COMPUTATIONS OF LAMINAR FLOW CONTROL ON SWEPT WINGS AS A COMPANION TO FLIGHT TEST RESEARCH

A Thesis

by

RICHARD GEORGE RHODES

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

December 2008

Major Subject: Aerospace Engineering

COMPUTATIONS OF LAMINAR FLOW CONTROL ON SWEPT WINGS AS A COMPANION TO FLIGHT TEST RESEARCH

A Thesis

by

RICHARD GEORGE RHODES

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Approved by:

Chair of Committee,	Helen Reed
Committee Members,	Jacques Richard
	Hamn-Ching Chen
Head of Department,	Helen Reed

December 2008

Major Subject: Aerospace Engineering

ABSTRACT

Computations of Laminar Flow Control on Swept Wings as a Companion to Flight Test Research. (December 2008)

Richard George Rhodes, B.S., Texas A&M University

Chair of Advisory Committee: Dr. Helen Reed

The high cost of energy has resulted in a renewed interest in the study of reducing skin-friction drag in aeronautical applications. Laminar Flow Control (LFC) refers to any technique which alters the basic-state flow-field to delay transition from laminar to turbulent flow. Achieving fully laminar flow over a civilian transport wing will significantly reduce drag and fuel costs while increasing range and performance.

Boundary-layer suction has proven to be an effective means of achieving laminar flow over an aircraft wing as demonstrated with the Northrop X-21 program; however, even with the savings in fuel, the high manufacturing and maintenance costs have discouraged the use of this technology. Recent work using three-dimensional (3-D) spanwise-periodic distributed roughness elements (DREs) has shown great promise as a means of controlling the crossflow instability responsible for transition over a swept wing without the need for a complex suction system.

The Texas A&M Flight Research Lab (FRL) is currently conducting flight tests of a laminar flow 30^{o} swept wing model (SWIFT) mounted vertically below the

port wing hard-point of a Cessna O-2A Skymaster. As a companion to flight experiments the current study is concerned with modeling the basic-state flow around the O-2 with the SWIFT model mounted on the port wing store pylon using a commercial Computational Fluid Dynamics (CFD) package. This basic-state solution serves two purposes: one is to analyze how the flow around the airplane affects the local flow around the model; secondly the basic-state boundary-layer profiles from the SWIFT model will be utilized by a boundary-layer stability code to perform analyses of the crossflow instability. The computational solution will be used to validate the flight test configuration as well as establish accurate chord-wise placement of the DRE's in order to achieve a minimum of 60% laminar flow over the low pressure side of the model.

Excellent agreement is obtained between flight data and the CFD basic-state solution justifying the use of the computational boundary-layer profiles for stability analysis. Once the boundary-layer profiles are properly extracted from the CFD data, a stability analysis predicts DRE placement which very closely matches flight test results.

DEDICATION

I would like to dedicate this work first to my parents who taught me the value of education. Without their stubborn, and at times frustrating, insistence that I live up to my potential, I surely would not be writing this paper. Second, this is dedicated to my wife Emily, who has also made this achievement possible with her steadfast support.

ACKNOWLEDGMENTS

The path to this work has brought me across many dear friends who have enriched my life and made me a better person. First, I must thank my advisor and committee chair Dr. Helen Reed for taking me on as a graduate student and for always making time to help me no matter how busy she might have been. I would also like to thank Dr. Jacques Richard, Dr. Ravi Srinivasan, and Dr. William Saric, for their expertise in the area of fluid dynamics and their patience which has been invaluable to me. Lastly, I would like to thank my fellow graduate students who have helped me along the way, mainly Andrew Carpenter, Shane Schouten, Ben Riley, Sriram Arasanapali, Sawan Suman, Sunny Jain, and Tucker Lavin.

LIST OF SYMBOLS

α_{ac}	Angle of attack of the aircraft
α_{sw}	Angle of attack of the SWIFT
α	Stream-wise wave-number
β_{ac}	Sideslip angle of the aircraft
β	Spanwise wave-number
$\Delta\theta$	Change in angle used by cosine boundary-layer clustering
ζ	Boundary-layer clustered domain
η	Similarity boundary-layer length scale
θ	Angle used in cosine boundary-layer clustering
λ_{SW}	Wing sweep angle
μm	Micrometer
ν	Dynamic viscosity
ρ	Density
σ	Constant multiplying the similarity boundary-layer length scale to adjust the grid truncation location
τ	Boundary-layer grid scaling, product of σ and η
φ	Flow-field quantity vector
φ'	Disturbance quantity vector
Φ	Basic-state quantity vector
ω	Disturbance frequency

b	Wing-span
C_P	Pressure coefficient
GB	Giga-byte
GHz	Giga-Hertz
KIAS	Knots Indicated Air Speed
k-ϵ	k-epsilon turbulence model
m	Meter
MB	Mega-byte
MHz	Mega-Hertz
mm	Millimeter
P	Pressure
R	Reynolds number based on similarity boundary-layer length scale
T	Temperature
u_e	Stream-wise velocity at the boundary-layer edge
u_∞	Stream-wise velocity at the far field
u,v,w	Orthogonal velocity components
x,y,z	Orthogonal coordinate system components

TABLE OF CONTENTS

CHAPTER Page

LIST OF FIGURES

Page

Page

Page

LIST OF TABLES

Page

CHAPTER I

INTRODUCTION

A. Background Motivation

Achieving fully laminar flow over a civilian transport wing will significantly reduce drag and fuel costs while increasing range and performance. Lachmann[1] showed that with oil at $27 per barrel (converted to 2008 dollars) fully laminar flow over the wings of a commercial airliner could reduce operating cost by 21.6%. Currently oil is hovering around $140 per barrel with fuel accounting for up to 72% of the cost of an airline ticket according to the Wall Street Journal[2]. Sturgeon[3] calculated that for a 200 passenger subsonic transport aircraft, a 28.2% reduction in fuel consumption would be possible with laminar flow over 75% of the wing chord surface and 65% of the tail chord surfaces, with today's fuel prices this would result in a direct cost per passenger reduction of 20.4% for American Airlines based on data in Table 1.

This thesis follows the style of the *AIAA Journal.*

Table 1 Estimated fuel cost per airline passenger[3].

Fill 'Er Up and Up

Fuel costs are eating up a huge portion of plane tickets. Here are estimates of current fares and fuel cost per passenger between New York and Los Angeles.

Airline	Route	Aircraft	Est. current average fare	Est. fuel cost per passenger	Share of ticket paying for fuel
American	JFK-LAX	767-200	$671	$488	72.7%
JetBlue	JFK-LGB	A320	$414	$292	70.5
Delta	JFK-LAX	737-800	$442	$299	67.6
United	EWR-LAX	757-200	$493	$314	63.7
Continental	EWR-LAX	737-800	$495	$293	59.2
United p.s.	JFK-LAX	757-200	$972	$520	53.5

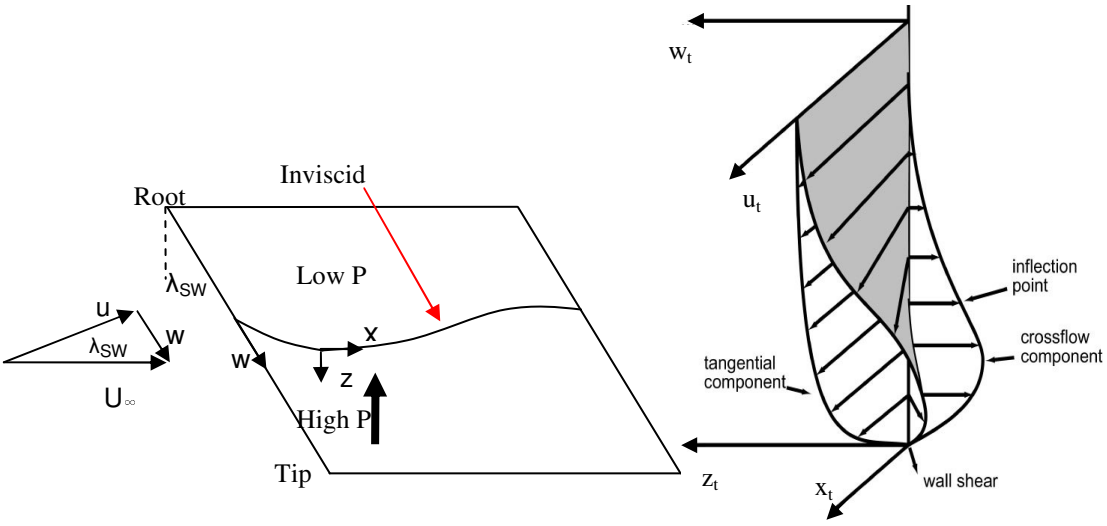
Source: WSJ estimates based on airline filings of Form 41 data at U.S. Department of Transportation.

Fares are based on fourth quarter 2007 averages for those routes, raised 10%. Per-passenger rates are based on average domestic load factor for each airline.

In the 1960's Northrop demonstrated with the X-21 program that a reduction in fuel consumption of around 200 gal./hr could be achieved with suction as a means of Laminar Flow Control (LFC). At today's jet fuel price of around four dollars per gallon, Northrop's X-21 would have saved approximately $800 per hour of cruise. The drawbacks of the LFC system as employed by Northrop's X-21 are the perceived high cost and problems with flight certification. Although recent advancements in manufacturing techniques of wing suction systems have shown the technology to be cost effective, air-framers remain hesitant to adopt such systems on the grounds that the risks still outweigh the benefits. Since laminar flow is extremely sensitive to atmospheric phenomena such as ice crystals, it is possible that during flight laminar flow might be interrupted resulting in an increase in fuel burn and

shorter range. In order for a laminar flow aircraft to be flight certified it must carry extra fuel to compensate for this uncertainty in range. The extra fuel adds weight therefore nullifying a large portion of the LFC benefits. Other well documented means of LFC include Natural Laminar Flow Control (NLFC) and Hybrid Laminar Flow Control (HLFC), with detailed reviews of both given by Green[4]. NLFC wings are designed to delay transition by accelerating the flow over the majority of the chord. The two primary limitations of NLFC, wing sweep and Reynolds number, are in direct competition. For aircraft of low wing sweep it is possible to maintain laminar flow at higher Reynolds number, as wing sweep is increased, the Reynolds number limit decreases. For a relatively straight wing the Reynolds number limits the aircraft size to be slightly larger than an Airbus A320 (Green[4]). NLFC cannot handle significant wing sweep angles due to the crossflow instability inherent in three dimensional (3-D) swept-wing boundary-layers. HLFC uses an NLFC airfoil modified with a flatter pressure distribution coupled with a wing leading-edge suction system to achieve roughly half-chord laminar flow over the upper wing surface. The advantages of HLFC compared with NLFC are that it can be applied to wings with greater sweep angles, larger aircraft, and higher Reynolds numbers. While the suction system employed by HLFC is not nearly as complex as that of a full chord suction system and therefore much cheaper to design, manufacture, and maintain, the savings in cost does not offset the risk associated with loss of laminarization while in flight.

Saric et al.[5] discuss in great detail the stability characteristics and transition modes of 3-D boundary-layers inherent in swept wing-flow. Figure 1a shows the path an inviscid streamline takes over the surface of a swept wing. The free-stream velocity is decomposed into a component normal to the leading edge and a component running along the leading edge such that as air approaches the wing, the velocity component normal to the leading edge goes to zero causing the streamline to run along the leading edge. This region is known as the attachment line. Aft of the attachment line at a given streamwise location an air molecule at an outboard spanwise station will have a lower velocity and thus higher pressure than at an inboard spanwise station, this force acts opposite the direction of span and curves the inviscid streamline back towards the freestream direction.



a) b) Source: Reibert et al.[6]

Fig. 1 Three dimensionality of a swept-wing boundary-layer. a) Inviscid streamline for a swept-wing. b) 3-D swept-wing boundary-layer profile.

Descending into the boundary-layer the spanwise pressure force which curves the inviscid streamline acts on lower momentum fluid causing a greater displacement resulting in a spanwise boundary-layer profile as shown in Fig. 1b. Here the coordinate system is aligned with the local inviscid streamline with the perpendicular direction identified as the crossflow direction. The instability arises from the boundary-layer inflection point in the crossflow velocity profile (Fig. 1b). This produces stationary crossflow vortices roughly aligned with the local inviscid streamline and serves as the primary instability in swept-wing flows. The crossflow vortices induce a slow mixing of the boundary-layer which leads to early nonlinear effects giving rise to additional inflection points and a high-frequency secondary instability which soon leads to transition. Radeztsky et al.[7] demonstrated the ultra-sensativity of the crossflow instability to spanwise periodic 3-D distributed roughness elements (DREs) applied near the attachment line. Saric et. al[8] took the next logical step and demonstrated a means of controlling the growth of crossflow vortices using DREs to excite spanwise modes with disturbance growth that peaks before inducing transition. The artificially induced control mode distorts the basic-state and stabilizes the naturally occurring unstable modes causing disturbance growth to peak prematurely; the end result is suppression of the crossflow instability and almost full-chord laminar flow. Saric et. al[8] demonstrated in wind tunnel testing at 2.4 million chord Reynolds number that DREs could effectively move the transition location from 71% chord well beyond the minimum pressure location and onto the trailing edge flap. This impressive result is accomplished using roughness

elements at a height of only 6 micro-meters (μm), the next step is to verify the DRE effectiveness in flight and at higher Reynolds numbers.

B. Objective

LFC is most beneficial for long-duration cruise flight where conditions do not vary significantly and laminar flow is seldom interrupted, therefore the application of the current research is to long-duration, high-altitude unmanned aerial vehicles (UAVs). The DREs provide a promising passive means of LFC in crossflow dominated swept-wing boundary-layers, possibly obviating the need for a complex wall suction system.

The crossflow instability has been well documented to be hyper sensitive to both micron-sized roughness and the free-stream vorticity inherent in even the best wind tunnels. In flight, the small-scale turbulence that interacts with a boundary-layer is absent, making flight testing the only possibility for careful research in this area.

The Texas A&M Flight Research Lab (FRL) is currently testing the effectiveness of the DREs applied to a swept wing test article (SWIFT) flown on a Cessna O-2A Skymaster (O-2). The O-2 is a top-wing aircraft with a twin-boom tail and both pusher and tractor propellers and is well suited for flight testing as it has mounting points at roughly half span of both the port and starboard wings which can accommodate a wide range of test equipment and models.

The SWIFT is mounted vertically beneath the port wing of the O-2 and is operated at chord Reynolds numbers between 7 million and 7.5 million. In order to

perform careful flight research, the local flow-field around the test article must be very accurately modeled and well resolved. Physical limitations restrict the placement of data probes during flight, therefore CFD analysis is needed to quantify the influence the airplane has on the SWIFT. Accurate placement of the DRE array is critical, thus necessitating a computational study which will model the growth of the instabilities to determine optimal chordwise placement of the DREs. This study is concerned with developing the computational model of the O-2 basic-state flow field and modeling the instability growth of the SWIFT boundary-layer.

C. Outline

Chapter II reviews the flight test configuration and methods employed by the FRL. Chapter III details the development of the computational mesh used to model the basic-state flow-field about the O-2 with the SWIFT. In order to construct a computational mesh, a solid model of the aircraft geometry must first be developed. Next, the computational mesh will be built based on this geometry using the GAMBIT[9]$_{TM}$ grid generation software which will serve as a baseline case for further grid refinement studies carried out in this work. Chapter IV is concerned with the application of the FLUENT[10]$_{TM}$ flow solver to the converged grid using upstream conditions corresponding to flight test points. Agreement of the CFD solution with flight test data is gauged based on comparison of pressure coefficient (C_P) curves from the suction side of the SWIFT model. Excellent agreement between CFD and flight data is shown in Chapter IV. As part of this study, the CFD model will be used to quantify the influence of the airplane on the SWIFT flow-field, and propose

possible changes to the flight test configuration including probe placement. In Chapter V the SWIFT boundary-layer profiles generated in FLUENT$_{TM}$ are used to perform a boundary-layer stability analysis using the NASA Langley code LASTRAC[11]$_{TM}$. The disturbance growth modeled using LASTRAC$_{TM}$ is used to compute the chord-wise neutral point where the DREs will be placed for LFC.

CHAPTER II

FLIGHT TEST CONFIGURATION

Flight testing is a challenging pursuit fraught with seemingly endless sources of measurement error and hidden regions of flow influence, the complexity of which rivals that of sub-atomic experimentation. By rigorously tracking all possible sources of error the FRL has assembled an excellent flight test configuration for the O-2 Skymaster outfitted with the SWIFT model. This chapter details the actual flight configuration and testing methods used by the FRL to conduct the LFC flight research that is modeled in this work.

A. Aircraft Configuration

The Cessna O-2A Skymaster is ideally suited for use as a test-bed for flight experiments. Born of the Vietnam War for use as a Forward Air Controller, the O-2 is fitted with mounting points underneath each wing for attachment of various weaponry configurations. The mounting points are essential for attaching flight test hardware; in this case the SWIFT model itself is attached underneath the port wing mounting point, while a probe measuring aircraft angle of attack (α_{ac}) and sideslip angle (β_{ac}) is attached to the starboard wing mount as shown in Fig. 2. This location of the air data boom turned out to yield erroneous measurements due to the influence of the aircraft; Chapter IV of this study addresses a solution to this issue.

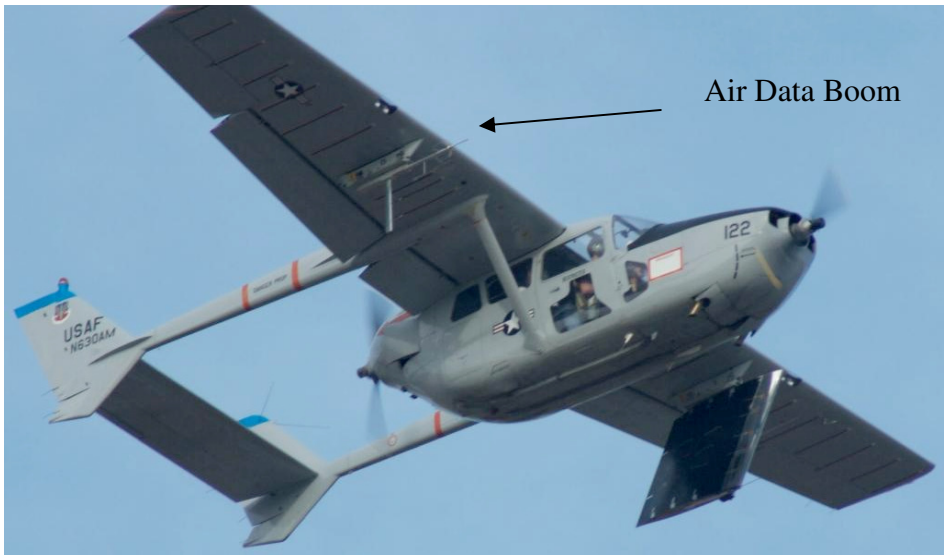


Fig. 2 Flight configuration of the O-2 Skymaster and SWIFT model.

B. SWIFT Model

The SWIFT model itself is a complex piece of test equipment that can be configured to collect a wide range of data on the suction (test) side where laminar flow is to be achieved. The streamwise placement of the SWIFT is such that any side-forces imparted on the O-2 act near the center of gravity reducing any moments on the airplane that may affect controllability. A Technical Review Board (TRB) and Safety Review Board (SRB) were conducted at the AFRL Wright Patterson AFB with the TRB summarized in two reports by Saric et al.[12,13] and the SRB summarized in Saric et al.[14]. In addition to the SRB, McKnight[15] detailed the design procedure required to keep the pitching moment of inertia within established limits. The model is hung such that the test side faces the fuselage of the aircraft to allow in-flight

observation of any contaminants that may impact the model and to allow flow visualization using instrumentation housed in the aircraft.

Infrared (IR) thermography is used in flight tests to identify turbulent and laminar zones and allow visualization of the DRE effectiveness. This technique has been proven to be an effective means of identifying laminar and turbulent flow regions in wind tunnel tests performed by Zuccher and Saric[16] as well as flight data from the NASA-DFRC study (e.g. Saric et al.[17]). Figure 3 shows a sample IR flight test image with the free-stream going from right to left. One can clearly see the stark contrast between laminar (cold) and turbulent (hot) zones due to the higher heat transfer rate characteristic of a turbulent boundary-layer. An appropriate temperature differential between the SWIFT surface and surrounding flow must be achieved to drive the heat transfer required for distinction between laminar and turbulent zones. In order to create this temperature differential in flight, the O-2 must be flown at a high altitude (10,500ft) for typically 30 minutes, where the SWIFT is chilled to a uniform temperature before the O-2 descends into warmer atmosphere where IR thermography can be conducted. During the ascent phase of the flight, temperature data is recorded as a function of altitude allowing the flight test engineer to identify possibly problematic altitudes where the IR thermography could be interrupted. In order to assure proper identification of turbulent zones with the IR camera, large roughness elements were placed on the leading edge to form turbulent wedges as shown in Fig. 3.

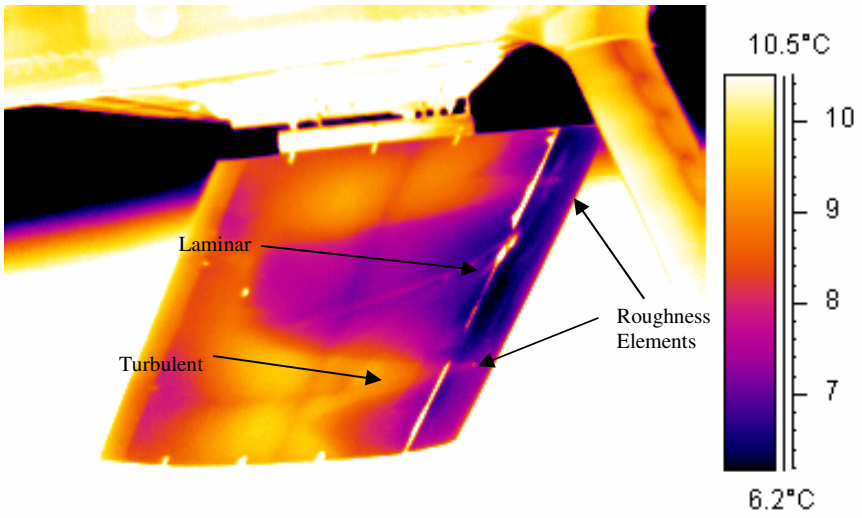


Fig. 3 IR Thermography showing turbulent and laminar zones while in flight[†].

C_P measurements are taken using two rows of pressure ports located as shown in Fig. 4. The ports are located at the 13 inch and 29 inch spanwise stations and are referred to in this study as the "inboard station" and "outboard station" respectively. The leading edge insert used for C_P measurements contains a tighter clustering of pressure ports to resolve the steep pressure gradient near the attachment line. The O-2 is not a high-speed aircraft and subsequently has a maximum level velocity of approximately 135 KIAS which does not allow achievement of the test condition chord Reynolds number of between 7 and 7.5 million. In order to achieve a higher flight chord Reynolds number, the FRL performs a high-speed descent in the O-2 reaching velocities up to 170 KIAS to achieve of the desired test condition. The high-speed descent also performs a second useful function, as the aircraft descends it

[†] IR photograph courtesy of Andrew Carpenter of the FRL

encounters warmer atmosphere which drives the heat transfer necessary in the previously outlined IR thermography.

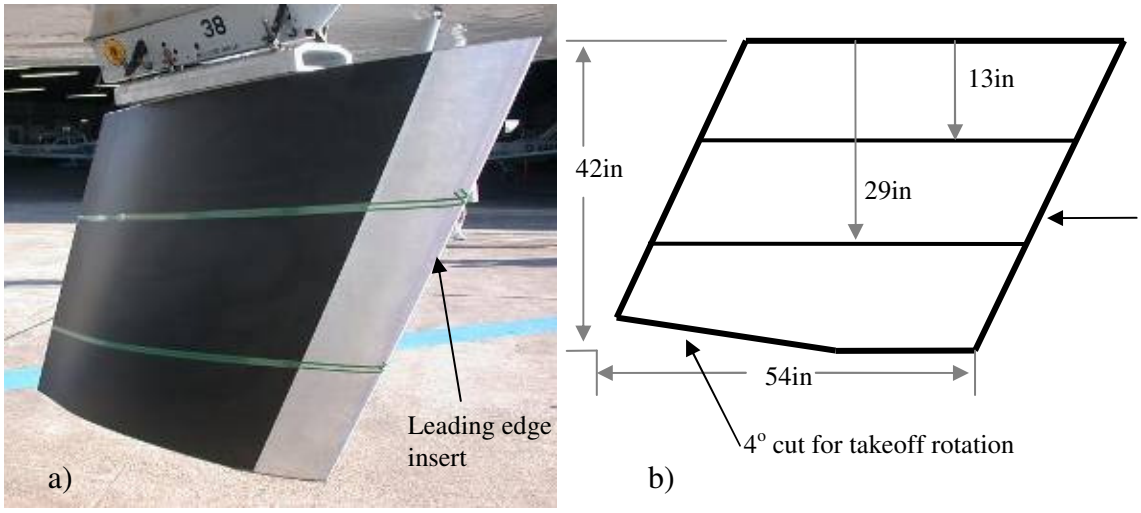


Fig. 4 SWIFT model as installed on the O-2. a) Photograph of SWIFT model configured for C_P measurement. b) Sketch of SWIFT geometry.

The FRL measured C_P in flight over a wide range of test conditions for the inboard and outboard stations, a sampling of these test conditions are provided in Table 2 and the corresponding flight C_P data are provided in Fig. 5. Note the leading edge clustering provided by the leading edge insert. The C_P data was used to generate boundary-layer profiles using a boundary-layer code utilizing an infinite-span swept-wing approximation. The boundary-layer profiles were used in a separate study to perform stability analysis for the inboard and outboard stations in order to obtain a low-order prediction of the neutral points for initial placement of the DREs while the current study was underway.

Table 2 Sampling of flight conditions used to capture C_P data.

TP	27	28	11	10	29	9	30	8	31	13
KTAS	179.7	173.1	160.9	163.9	166.5	166.4	169.2	172.9	158.	158.2
Rec/million	7.15	7.13	7.20	7.16	7.03	7.14	7.36	7.13	7.08	7.38
Q (psid)	0.60	0.58	0.54	0.54	0.55	0.55	0.58	0.57	0.52	0.55
P (psia)	10.98	11.42	12.18	11.93	11.65	11.75	11.93	11.34	12.22	12.93
T (deg. C)	1.31	2.02	-1.43	-0.69	0.72	-0.33	-0.13	0.61	-1.06	1.52
Alpha 5HP (deg.)	**-0.91**	**-0.84**	**-0.49**	**-0.58**	**-0.58**	**-0.58**	**-0.64**	**-0.76**	**-0.47**	**-0.62**
Beta 5HP (deg.)	4.69	4.26	4.17	3.75	3.69	3.24	3.18	2.71	2.61	2.19
AoA 5HP (deg.)	**-4.69**	**-4.26**	**-4.17**	**-3.75**	**-3.69**	**-3.24**	**-3.18**	**-2.71**	**-2.61**	**-2.19**

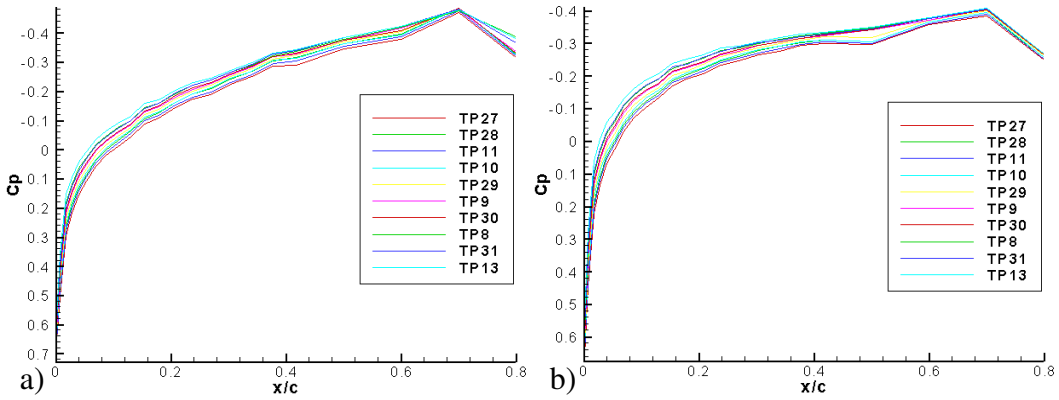


Fig. 5 C_P data taken at test points shown in Table 2. a) Inboard station. b)Outboard Station.

During the high-speed descent maneuver, the pilot must take great care in maintaining the desired flight conditions. The O-2 is yawed in order to control the SWIFT angle of attack (α_{SW}), similarly the O-2 angle of attack (α_{ac}) changes the effective sweep angle (λ_{SW}) of the SWIFT. Positive α_{ac} serves to bring the base of the SWIFT forward thus reducing the effective sweep angle; negative α_{ac} increases the effective sweep angle. The sideslip convention used by the FRL is such that positive aircraft side-slip angle (β_{ac}) is defined as starboard wing forward, therefore positive β_{ac} corresponds to negative α_{sw}. Figure 6 illustrates the relationship between aircraft

sideslip and SWIFT angle of attack. The sign convention used for α_{ac} is the standard convention where positive α_{ac} corresponds to aircraft nose-up.

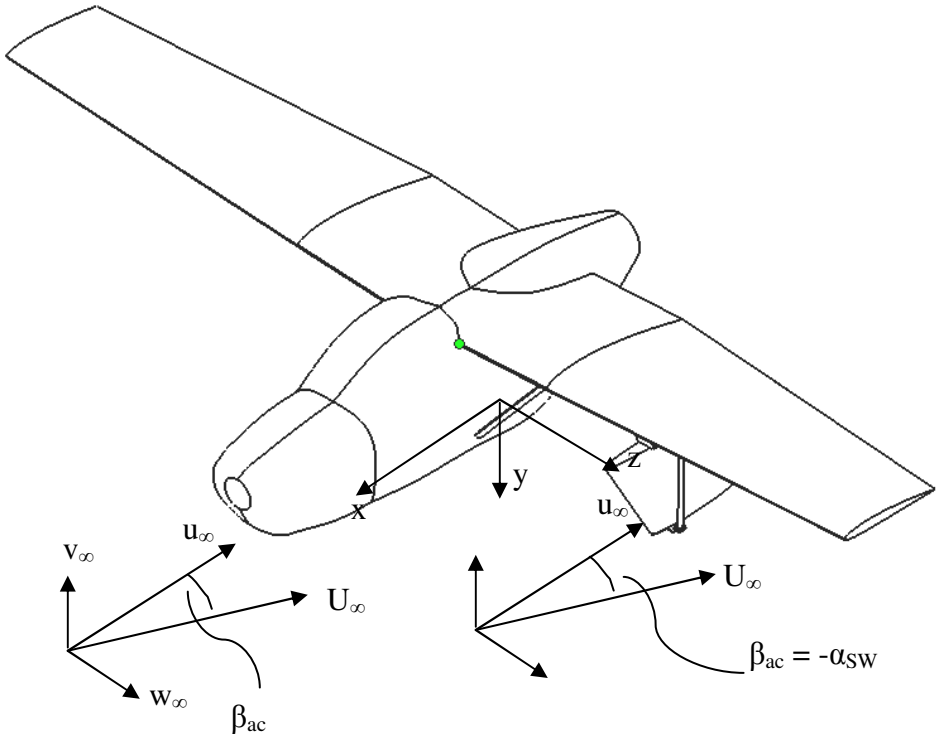


Fig. 6 Alpha-Beta relationship between the O-2 and SWIFT.

This study is concerned with the computational modeling of the test configuration detailed in this chapter. To construct a model suitable for CFD simulation, simplifications to the aircraft and SWIFT will have to be made, however care must be taken to capture the critical physics of the problem. To this end, C_P measurements from the CFD solution will be compared with flight test data to gauge how accurately the experimental flow-field is being modeled. Contour plots will also be used to verify that the computational model is behaving as one would expect.

Once a suitable solution has been obtained, boundary-layer profiles will be extracted from lines along the two rows of pressure ports on the low-pressure (test) side of the SWIFT. These profiles will be used in conjunction with a boundary-layer stability analysis software package to predict optimum placement of the DREs.

CHAPTER III

DEVELOPMENT OF THE COMPUTATIONAL MESH

The practice of CFD is often likened to art in that the user's experience and intuition is just as critical as his or her understanding of the theory. This analogy especially holds true when building a complex three-dimensional mesh, where the criteria for constructing a stable and accurate mesh are not well quantified. For the initial mesh, the engineer is forced to rely on past experience with the type of problem at hand, whether it is personal experience, or experience gleaned from the many "rules of thumb" established in the computational literature. Following the initial mesh, further iterations are established based on more rigorous trade studies performed for mesh variables. The permutations of grid configurations can be endless, and again the engineer must act more as an artist to navigate this impossible labyrinth and achieve grid convergence.

A. Solid Model

To model the basic-state, a simplified solid model of the O-2 with the SWIFT model attached was created using Solidworks[18]$_{TM}$. The baseline Solidworks$_{TM}$ model of the O-2 with the SWIFT attached was obtained from the FRL, however many iterations were required to achieve a geometry database suitable for CFD. The region under the port wing including the SWIFT and its mount is considered to be the most critical region and was completely reconstructed using measurements taken directly from the aircraft. It was assumed a priori that the horizontal and vertical tail surfaces,

starboard tail boom, and starboard strut were located far enough from the SWIFT to have negligible influence and were therefore discarded from the model altogether. Later, agreement with flight data will show this assumption to be valid. The final configuration of the geometry database to be used for grid generation is shown in Fig. 7.

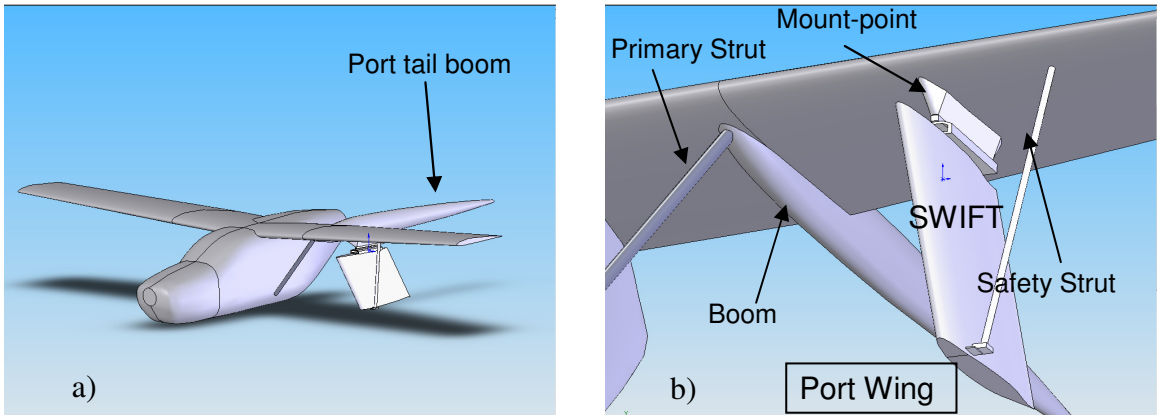


Fig. 7 Solid model used to generate a computational mesh. a) Entire geometry. b) Close-up of the SWIFT.

B. Meshing Technique

The computational mesh was constructed using the geometry database as imported from Solidworks$_{TM}$. The ANSYS grid generator GAMBIT$_{TM}$ is used in this study to construct the computational mesh primarily because both GAMBIT$_{TM}$ and its associated flow solver FLUENT$_{TM}$ were readily available and have been extensively tested. GAMBIT$_{TM}$ has advanced algorithms for constructing an unstructured quad or tetrahedral mesh in two and three dimensions and can also construct structured algebraic grids. The standard version of GAMBIT$_{TM}$ however

cannot perform elliptical grid generation or be used to construct a Chimera type embedded mesh.

For creation of the mesh, a Dell Precision 470n workstation running an open-source version of the Linux Operating system was used. The 470n workstation was used for solution of some of the smaller computational meshes using FLUENT$_{TM}$, however the mesh quickly out-grew the memory capabilities of the workstation and had to be solved using much larger supercomputers managed by the Texas A&M Supercomputing Center. The two supercomputer systems used in this study are known as Cosmos, an SGI ALTIX 3700 system, and Hydra, an IBM P5-575 cluster. The hardware specifications of all three systems are provided in Table 3.

Table 3 Supercomputer hardware specifications.

	Dell Precision 470n	**Cosmos**	**Hydra**
Number of	2	128	32
Processor Type	64 bit Intel Xeon Dual Core 2.8GHz	64 bit Intel Itanium-2 1.3GHz	IBM Power-5 Dual-core 1.9GHz
Processor Cache	2MB per processor	256KB L2, 3MB L3	1.9MB L2, 36MB L3
FSB	800MHz	200/400MHz	Distributed Fabric
RAM	16GB PC2-3200 DIMM	256GB shared	1080GB
Hard Drive	160GB SCSI	4x36GB system, 10TB expansion	10 TB
Operating System	64 bit Ubuntu 6.06	IA-64 Linux	64bit AIX 5L (5.3)

While an unstructured mesh is invaluable for modeling complex 3-D geometries, it is almost powerless to resolve a 3-D boundary-layer without incurring

an enormous cell count. This inadequacy arises from the unstructured grid generator's inability to differentiate cell growth along different axes, where as with a structured quad mesh, one can control the cell x, y, and z dimensions independently. Fig. 8 illustrates in 2-D the advantage of using a Structured mesh to resolve a boundary-layer, the case of a 3-D mesh would show an even greater advantage as the cell count in the third dimension is multiplicative.

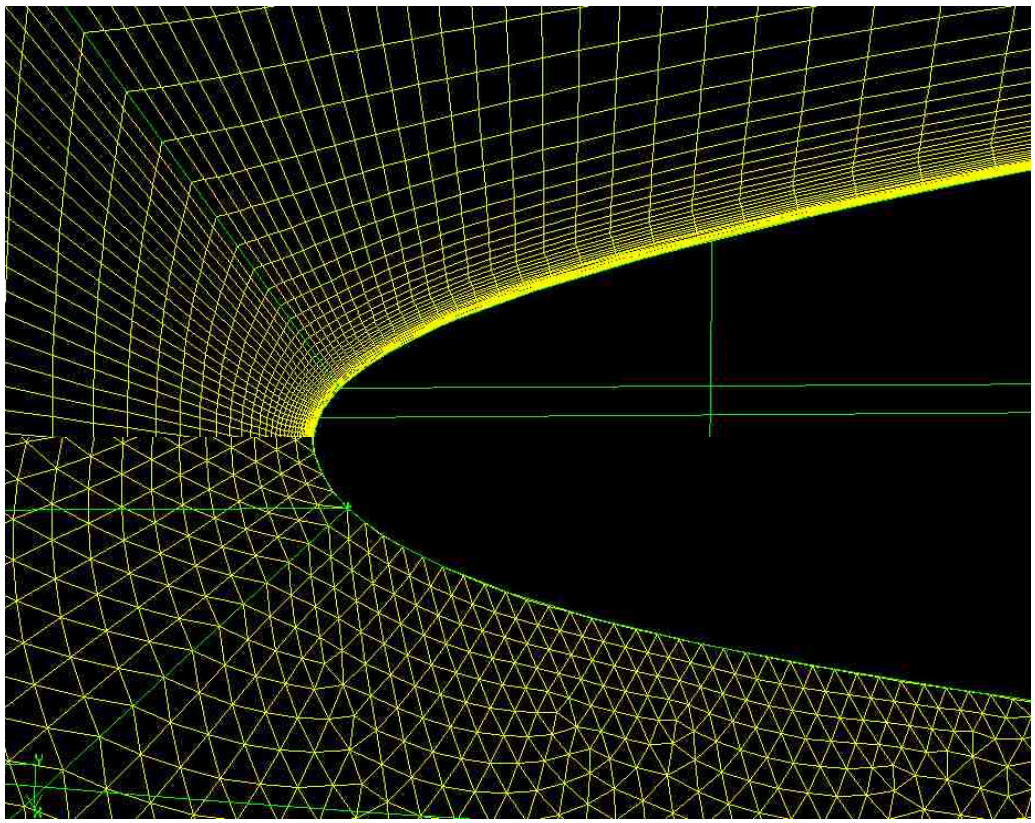


Fig. 8 Structured vs. unstructured mesh for boundary-layer resolution.

The upper structured mesh in Fig. 8 has wall-normal cell dimension of 0.0005 inches and has a total cell count of 16,750 compared to the lower

unstructured mesh with a wall-normal cell dimension of 0.1 inches and a total cell count of well over 30,000. The boundary-layer in this study is on the order of a hundredth of an inch thick in some areas rendering the unstructured mesh completely inadequate for a viscous solution. To capture the boundary-layer on the SWIFT, yet retain the flexibility of an unstructured mesh in geometrically complex regions, a hybrid approach must be taken where the swift is "wrapped" in a structured C-grid that interfaces with a larger unstructured grid fit over the rest of the airplane. Figure 9 shows the different structured and unstructured regions constituting the mesh of the O-2 with the SWIFT model attached.

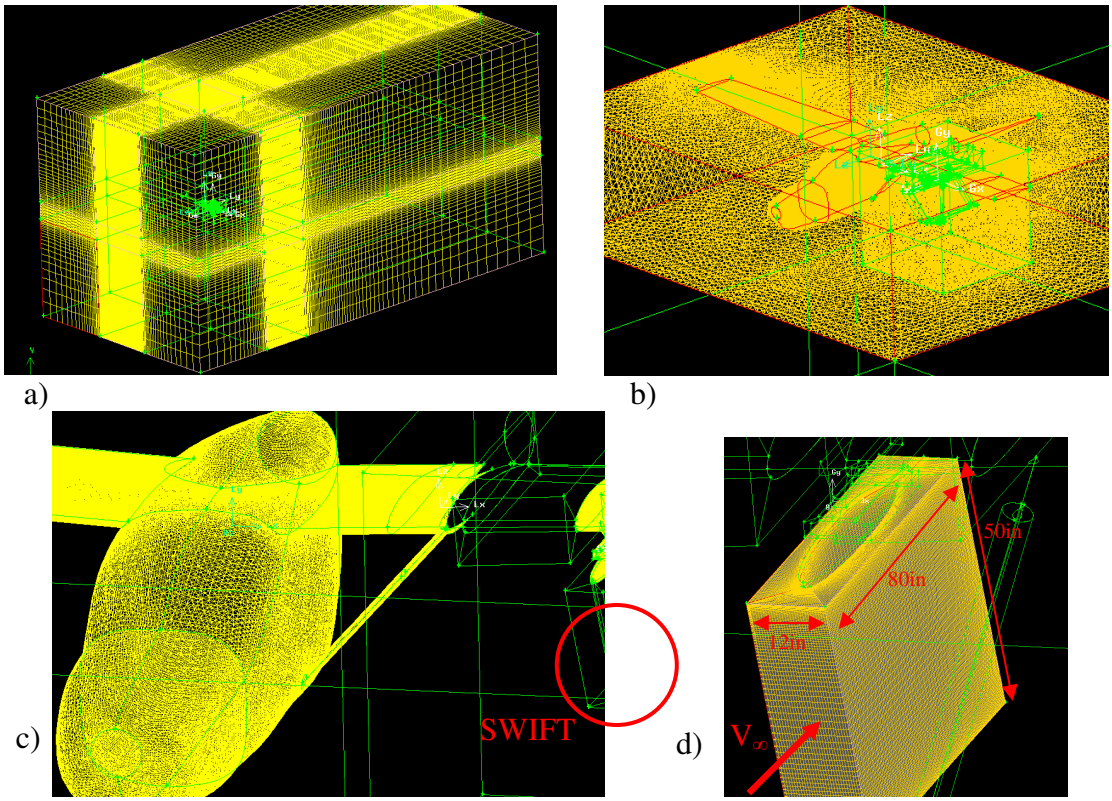


Fig. 9 Grid constructed around the O-2 and SWIFT. a) Outer structured domain. b) Intermediate unstructured region. c) Unstructured mesh applied to fuselage. d) Structured region around the SWIFT.

For the extreme outer region of the grid, a structured mesh was employed as shown in Fig. 9a allowing excellent wake clustering behind the O-2 at a very small cost of slightly over 1 million cells. A separate cube was fit over the region immediately around the O-2 as shown in Fig. 9b, this allowed containment of the unstructured mesh applied to the surface of the O-2. A second cube is fit around the SWIFT, safety strut, and port wing. This region is also unstructured but is partitioned to allow an additional degree of control over the growth of the unstructured mesh in the region closest to the SWIFT. The structured region fit over the majority of the SWIFT span is shown in Fig. 9d and shows the manner in which the conformal mesh was applied. A small region at the root and tip of the SWIFT had to be excluded from the structured region to allow proper mapping of the conformal mesh. The root and tip of the SWIFT that are excluded from the structured region are meshed with unstructured cells in a similar fashion as the O-2.

To obtain an initial solution, the computational domain was initially set to be a cube with dimensions eight times the O-2 wingspan (~443in) in the stream-wise direction and four times the wingspan in the other two principal directions. Velocity inlet boundary conditions were used for all outer faces of the domain except the back face, which was specified as a pressure outlet. No-slip wall conditions were applied to all solid surfaces, and grid interface conditions were used to pass information between structured and unstructured regions. Table 4 shows the pertinent solution parameters specified in FLUENT$_{TM}$ for the initial case.

Table 4 FLUENT$_{TM}$ Parameters used in initial solution.

Parameter	Value
Solver	Steady-State, Incompressible
Momentum Discretization	1st Order
Pressure Discretization	1st Order
Pressure-Velocity Coupling	SIMPLE Algorithm
Viscous Model	Laminar
Convergence Criteria	1.E-5

To determine an appropriate computational domain size, the velocity field was used to visualize the downstream influence of the airplane and model by using contour plots. Figure 10 shows a top view of the computational domain filled with contours of velocity magnitude, distance units are given in "b", the wingspan of the O-2, velocity units are given in feet per second.

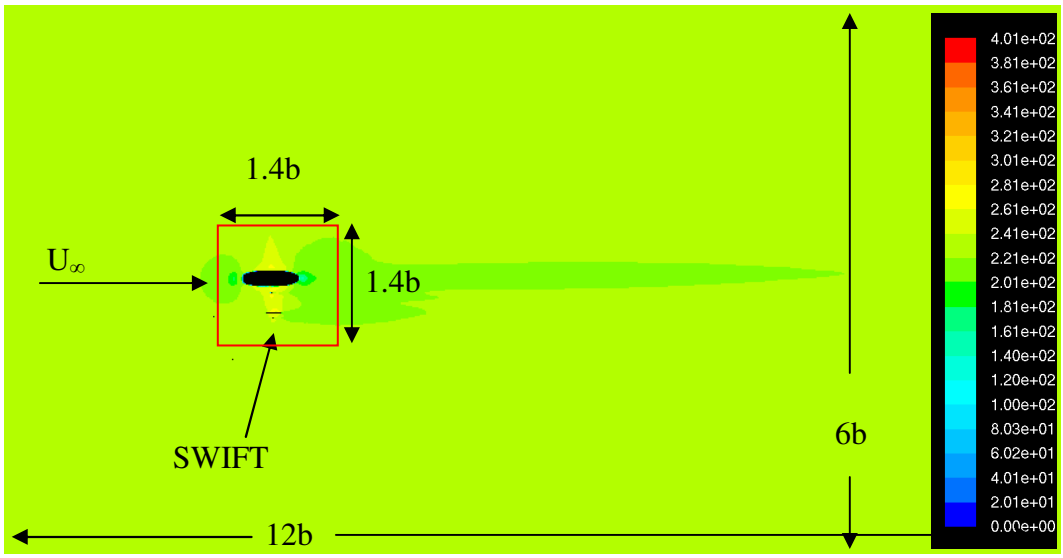


Fig. 10 Velocity contour in ft/s around the O-2 and SWIFT over the entire domain.

One can clearly see that the wake of the aircraft and SWIFT are fully captured in the domain. The red box in Fig. 10 approximates the boundary between the outer structured region and the unstructured region, everything within this box is unstructured with the exception of the SWIFT, while the mesh outside of this box is structured. One can clearly see the contours of velocity magnitude pass continuously through the grid interface. Since the pressure distribution over the test side of the SWIFT showed no change when the domain was grown to its current size, the author chose to use the larger domain because the cell increase was minimal and it correctly captured the wake of the O-2 allowing faster solution convergence. A grid convergence study was performed on the mesh using C_P as a basis for comparison; Fig. 11 shows C_P data for three levels of grid refinement with "Grid 1" being the least refined and "Grid 3" being the most refined.

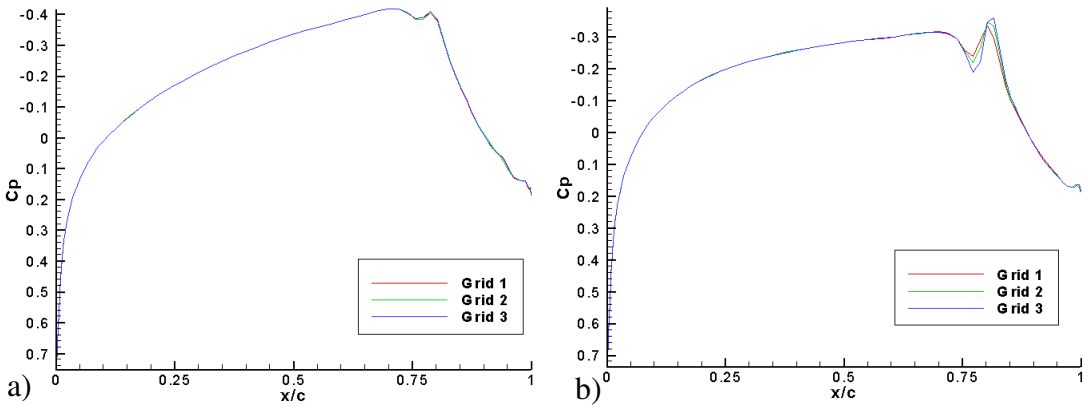


Fig. 11 C_P plots of grid convergence study. a) Inboard station. b) Outboard station.

One can see from Fig. 11 that there is some small deviation aft of the minimum pressure location for the outboard station which does not seem to diminish

with grid refinement leading to the hypothesis that this is a laminar separation region. In Chapter IV of this study the use of turbulence modeling proves this "bump" to be a laminar separation artifact. Since the end goal of this study is to generate accurate SWIFT boundary-layer data to be used for stability analysis performed near the leading edge, the behavior of the solution aft of the minimum pressure location is not considered. The conclusion of the grid convergence study is that the pressure coefficient (C_P) curves on the test-side (suction) of the SWIFT showed no change for the refined meshes, thus the mesh parameters from the "Grid 1" column in Table 5 are retained for this study. This result is not surprising considering the first mesh was "over-designed" with a fine mesh as cell count was not a great concern given the large amount of system memory on the available computational resources.

Table 5 Grid parameters used in convergence study corresponding to Fig. 11.

Grid Parameter	Grid 1	Grid 2	Grid 3
Outer Domain Streamwise Length	5316in	5316in	5316in
Outer Domain Height and Width	2658in	2658in	2658in
SWIFT Zone Number of Normal Points	50	60	70
SWIFT Zone BL Mesh Growth Rate	1.3	1.2	1.1
SWIFT Zone Number of Chord-wise Points	160	200	250
SWIFT Zone Spanwise Number of Points	78	120	140
SWIFT Unstructured Surface Mes Size	0.2 in	0.15 in	0.15 in
Safety Strut Unstructured Surface Mesh Size	0.2 in	0.15 in	0.15 in
Port Wing Surface Mesh Size (min/max)	0.3 in, 0.6 in	0.2 in, 0.6 in	0.1 in, 0.6 in
Airplane Surface Mesh Size (min/max)	1 in, 4 in	0.5 in, 3 in	0.5 in, 3 in
Port Wing Zone Unstructured Mesh (growth rate/max size)	1.2 , 8in	1.1 , 6in	1.05 , 6in
Airplane Zone Unstructured Mesh (growth rate/max size)	1.3, 12in	1.2, 10in	1.15, 10in
Outer Structured Mesh Range	10in-120in	10in-100in	8in-80in
Total Cell Count (million)	19.3	22.1	22.4

C. Trade Study

After completing a grid convergence study on the baseline mesh, there remained two possible major sources of influence on the SWIFT to be eliminated. The first is the propeller, which was initially omitted in this study, and the second is the safety strut attached to the SWIFT which has some nominal incidence that is difficult to measure.

Since the O-2 has both a pusher and tractor propeller there is a possibility that the upstream (tractor) propeller has some influence on the SWIFT. To eliminate this possible source of error, a CFD investigation was performed in which the spinning front propeller was modeled using a sliding mesh zone with an unsteady time-

dependent solver. Since no data could be found on the propeller airfoil geometry, a solid model of the propeller was constructed using the plan-form shape as measured directly from the actual propeller with a NACA 0012 airfoil approximating the cross-section. The hub of the propeller was eliminated due to its complex shape and relatively small contribution to the propeller flow-field. Figure 12 shows the simplified geometry used in this model and Table 6 gives the dimensions in greater detail.

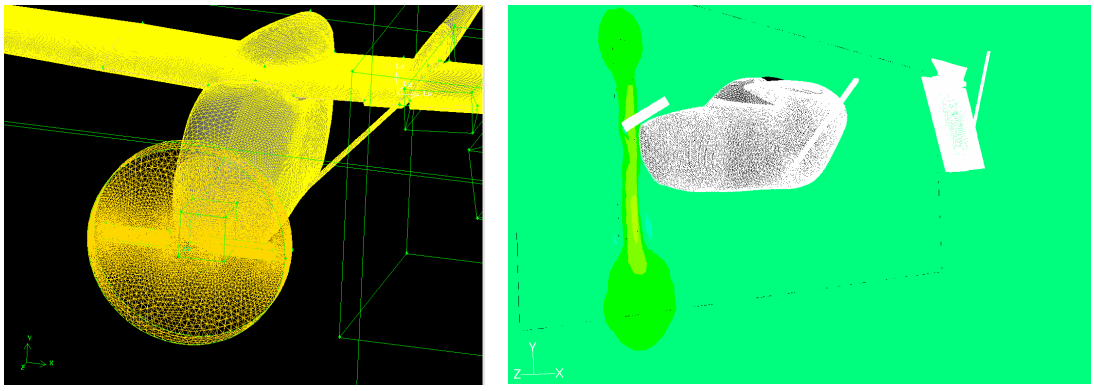
a) b)

Fig. 12 Propeller geometry compared with approximation. a) Plan-form approximation. b) Airfoil section approximation.

Table 6 Propeller model dimensions.

Airfoil	NACA 0012
Root Chord	4.5 in
Tip Chord	3.75 in
Root Span	4.875 in
Root Incidence	40°
Tip Span	33.125 in
Tip Incidence	23°

The equation for a NACA 4-series airfoil was used to generate coordinates constituting the cross-sections at the root and tip of the propeller. Instead of creating the propeller in Solidworks$_{TM}$ as with the O-2 and SWIFT, the airfoil coordinates at the root and tip or the propeller were imported directly into GAMBIT$_{TM}$ using the turbo machinery sub-pad. The automated algorithms within GAMBIT$_{TM}$ then constructed both propeller blades based on the airfoil sections. The propeller was encased in a disk filled with unstructured elements which physically rotated for every time-step advancement. The boundaries between the rotating disk and O-2 mesh were handled using a grid interface condition which allowed the propeller to be a sliding mesh zone. The mesh applied to the propeller as well as a contour plot of pressure superposed on the mesh is shown in Fig. 13. It is apparent from the contour plot that the solution smoothly passes through the sliding grid interface encasing the propeller.



a) b)

Fig. 13 Propeller mesh used in unsteady calculations. a) Mesh before solution. b) Contour plot of total pressure superposed over grid.

The unsteady rotating propeller mesh was solved using FLUENT$_{TM}$ with the time-step advancing until the C_P curves on the test side of the SWIFT ceased to change, effectively converging the solution in the temporal domain. Although calculations were performed up to the 400th time-step, the solution was effectively converged after approximately 250 time-steps, The converged C_P data for the solution including the propeller is compared with the steady-state solution without the propeller in Fig. 14. Table 7 provides the solver configuration used in the unsteady solution.

Table 7 Solver options specified in the unsteady solution.

Option	Value
Viscous Model	Laminar
Velocity Formulation	Absolute
Time	Unsteady
Time-step (Fixed)	5.e-3 s
P-V Coupling	SIMPLEC
Pressure Discretization	Standard
Momentum Discretization	1st order
Propeller Zone RPM	2200

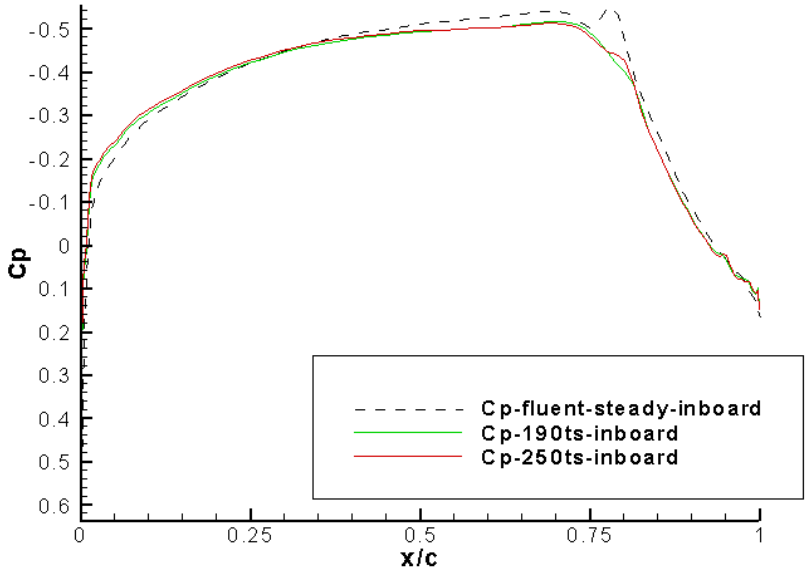


Fig. 14 C_P data taken at the inboard station from the converged unsteady propeller solution.

The difference in the C_P curves between the steady and unsteady CFD solutions is very small. Furthermore the FRL performed flight tests in which the front propeller RPM was varied from 1700 to 2300. The C_P plots from this test are shown in Fig. 15, the small difference in C_P is attributed to the slightly different free-stream flow angles between the two flight conditions. For the 1700 RPM case, α_{ac} = 3.33 and β_{ac} = 4.81, and for the 2300 RPM case, α_{ac} = 3.63 and β_{ac} = 4.58. Based on this flight observation it is assumed that the small difference in computational C_p between steady and unsteady solutions arises from complications introduced by the large number of grid interfaces required for the rotating propeller mesh. The propeller is therefore assumed to have negligible affect on the pressure distribution over the SWIFT and is omitted from further calculations.

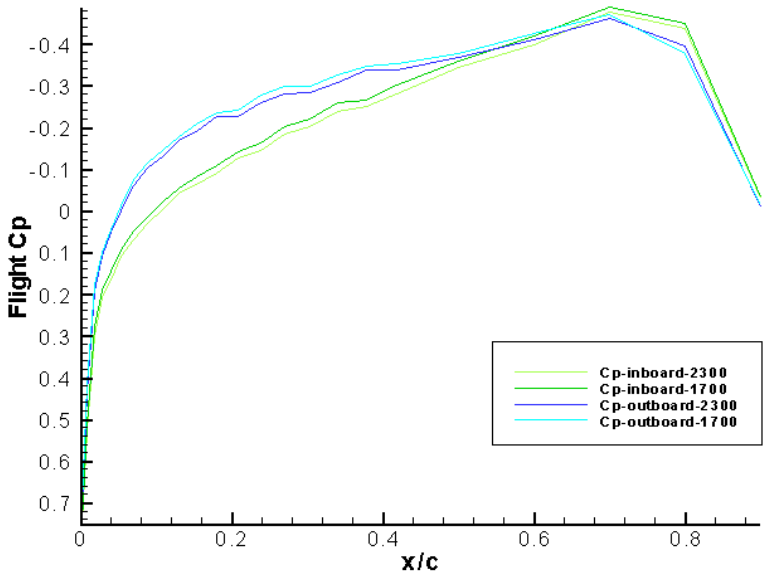


Fig. 15 Flight C_P data taken at the inboard and outboard stations for front propeller RPM of 1700 and 2300.

The second likely source of influence to be investigated is the safety strut attached to the high-pressure side of the SWIFT. The strut cross-section is effectively a symmetric airfoil, although it is not based on any established airfoil shape. The main concern of the author was the incidence of the strut and how the wake might impact the high-pressure side of the SWIFT possibly influencing the test side. Figure 16 shows the safety strut as attached in the flight configuration, and Fig. 17 shows the strut cross section.

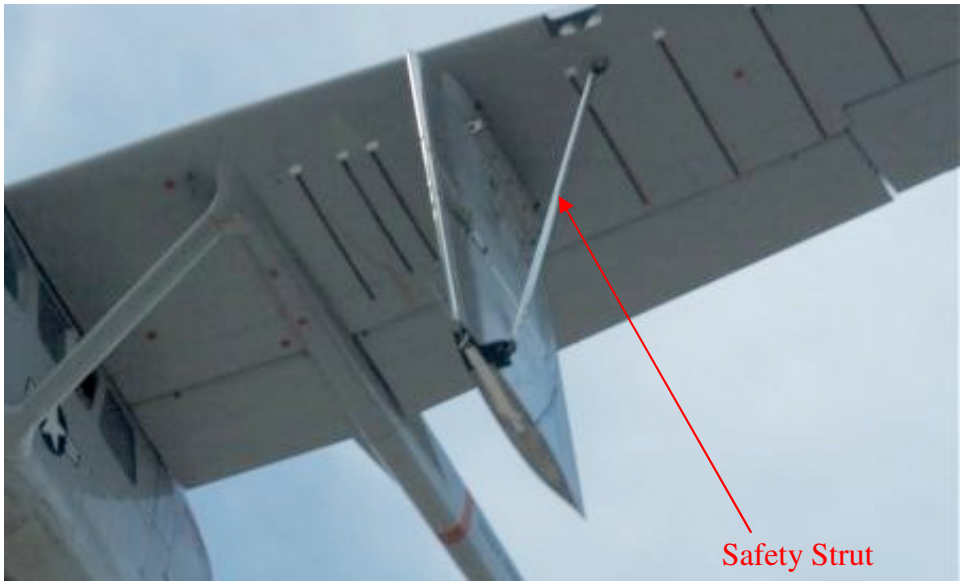


Fig. 16 Safety strut as attached to the SWIFT.

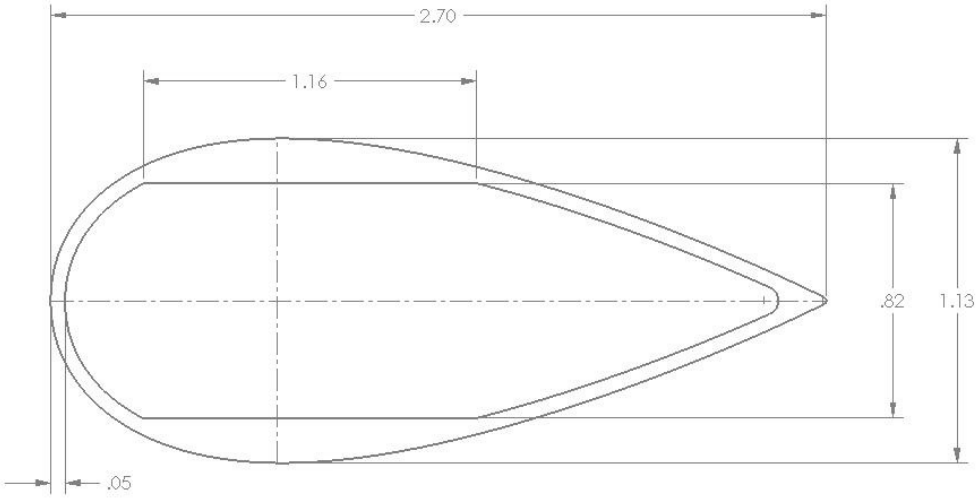


Fig. 17 Diagram of safety strut cross-section in inches.

The incidence of the strut is difficult to establish since the thickness to chord ratio is on the order of unity, therefore multiple meshes were constructed with the strut rotated at incidence angles between plus and minus 20^{o} based on a nominal incidence that is approximately zero. Any possible incidence angle error would be much smaller than 20^{o}, however a large number was chosen to provide an extreme

worst case scenario. If an influence was found, then smaller angles would be evaluated to converge on the correct angle. Figure 18 shows C_P data taken with the strut at the assumed nominal position compared to the +/-20^o positions and shows conclusively that the strut has effectively no influence on the test side of the SWIFT.

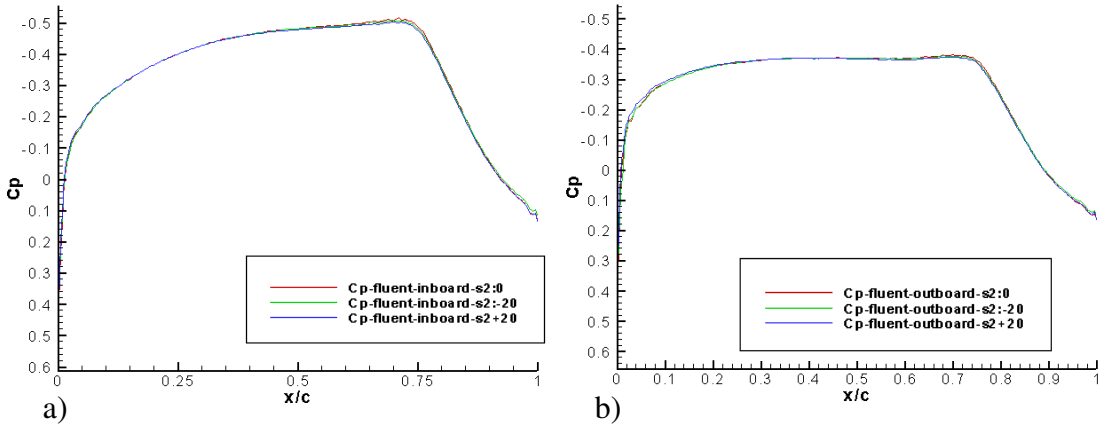


Fig. 18 C_P data comparing the incidence of the strut varied +/- 20^o. a) Inboard station. b) Outboard station.

D. Final Configuration

The converged mesh is based on the original geometry established at the beginning of this chapter. The propeller has been omitted allowing a steady state solution and the incidence of the safety strut is left assumed to be zero. The maximum cell skewness of all unstructured regions is below 0.95 allowing efficient solver convergence without the need for a skewness correction algorithm. The total cell count is approximately 19 million requiring at least 24GB of ram and 4 processors in parallel to solve.

CHAPTER IV

FLIGHT CONDITIONS AND SOLVER OPTIONS

To accurately model a flight test, one must take into account the discrepancy between local measurements and upstream conditions; mainly that what is considered a free-stream quantity in experiment is actually a local quantity in the area of a data probe. If a discrepancy exists between the actual free-stream quantity and the local quantity at the probe, then this must be correctly implemented in the computational model. Furthermore in order to obtain an accurate solution the correct flow solver options must be specified. FLUENT$_{TM}$ provides a plethora of solver options and discretization schemes making selection of the correct configuration as critical as constructing an appropriate grid. The options can be somewhat narrowed by taking into account the characteristics of the problem at hand. For example the Mach number for the flight tests to be modeled is around 0.2, safely within the incompressible limit, therefore configuring the solver for a compressible solution is not necessary.

The flight conditions to be evaluated in this study are test points (TP) 27 and 31 shown in Table 2. Neutral point calculations were conducted in a separate study with basic-state boundary-layer profiles generated with a boundary-layer code using an infinite-span swept-wing approximation along with the C_P and free-stream data recorded in flight. These early calculations along with flight tests have shown TP31 to be a good candidate for testing the effectiveness of the DREs. TP27 was selected

to evaluate the DRE effectiveness at a more negative angle of attack than that in TP31. Since in this study calculations are conducted using basic-state boundary-layer profiles generated by a full Navier-Stokes code (FLUENT$_{TM}$), it is expected that the neutral points predicted will be more accurate than those based off of the profiles generated with the boundary-layer code.

A. Flow Angularity

Initially all free-stream measurements for the O-2 were taken from the air data boom attached under the starboard wing mount. After the CFD solution failed to match the flight C_P data by a large margin, the FRL attached a free-stream data probe to the high-pressure (non-test) side of the SWIFT with the goal of measuring the local flow experienced by the SWIFT. The probe used is a 5-hole probe capable of measuring the incoming flow angle and velocity. Figure 19 shows the O-2 configured with the additional data probe mounted under the port wing as well as the original air data boom mounted under the starboard wing.

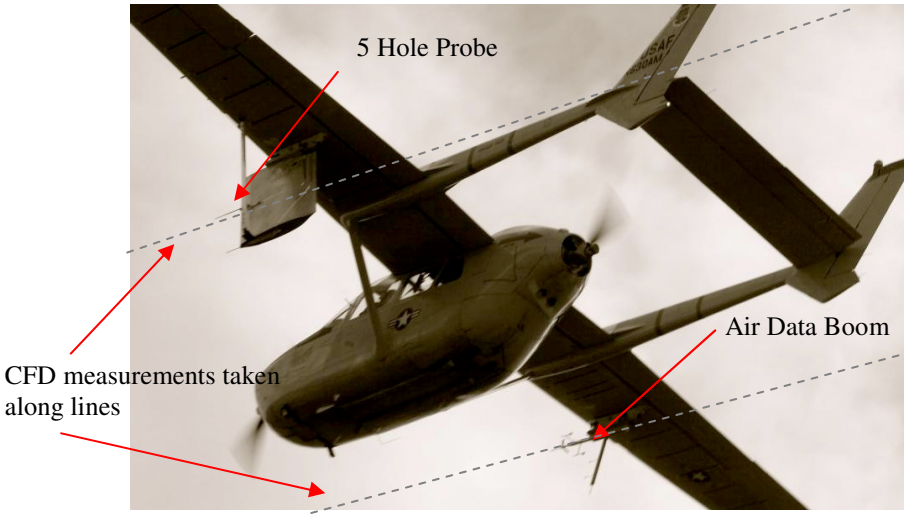


Fig. 19 Air data boom installation on the O-2 with additional 5-hole probe.

The data taken from the 5-hole probe showed that the SWIFT was experiencing an angle of attack on the order of 2^{o} different from that measured by the air data boom on the starboard side. Upon observing this discrepancy a CFD investigation was performed to investigate the influence of the O-2 on the angle of attack experienced by the SWIFT. To visualize the influence of the O-2 fuselage on the flow angle, a CFD analysis was performed for the case of zero sideslip and zero angle of attack. Measuring the flow angle in the CFD solution along lines shown in Fig. 19 which extend to the upstream and downstream limits of the domain along the air-data boom and 5-hole probe yields a plot as shown in Fig. 20.

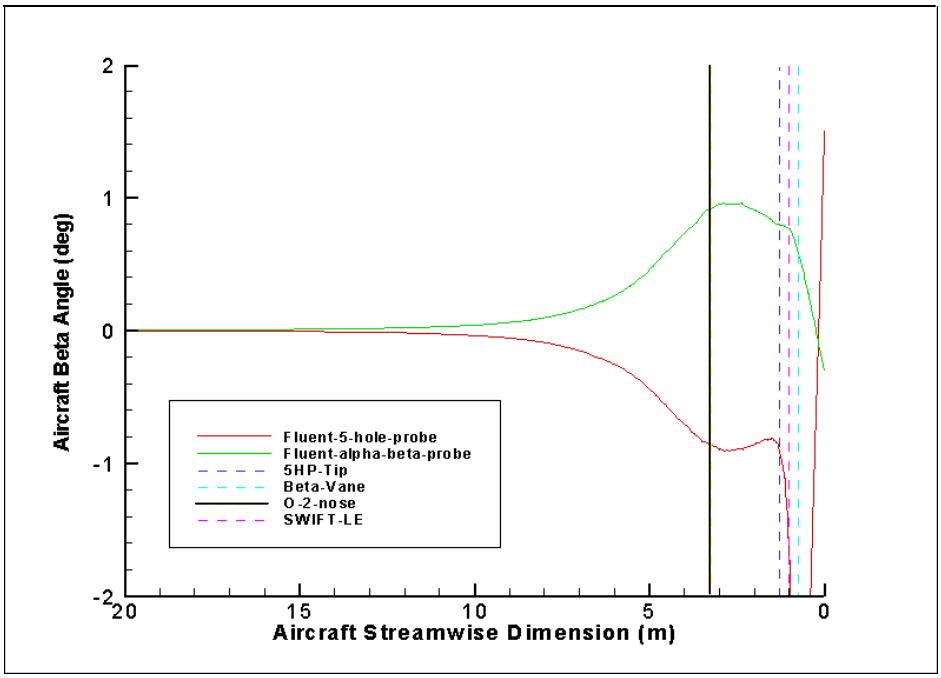


Fig. 20 Flow angularity between the port and starboard wing mount locations.

Looking at Fig. 20, the flow goes from left to right and vertical lines show the locations of critical points along the O-2. One can clearly see the large amount of curvature in the flow induced by the fuselage. Considering the solid red line (port side) in Fig. 20, one can see the upstream influence of the SWIFT. Using this data, the optimal location of the 5-hole probe tip relative to the SWIFT was determined to be half-span, 25 inches upstream of the leading edge and 5.5 inches out from the pressure side. Since the influence of the aircraft is shown to extend well over 10m upstream, it was determined to be physically impossible to construct a probe long enough to measure the correct free-stream flow angle. Furthermore for the purposes of flight testing it is sufficient to know only the flight conditions experienced by the

SWIFT making measurement of the true free-stream unnecessary. Fig. 21 shows the five-hole probe installed in its final configuration.

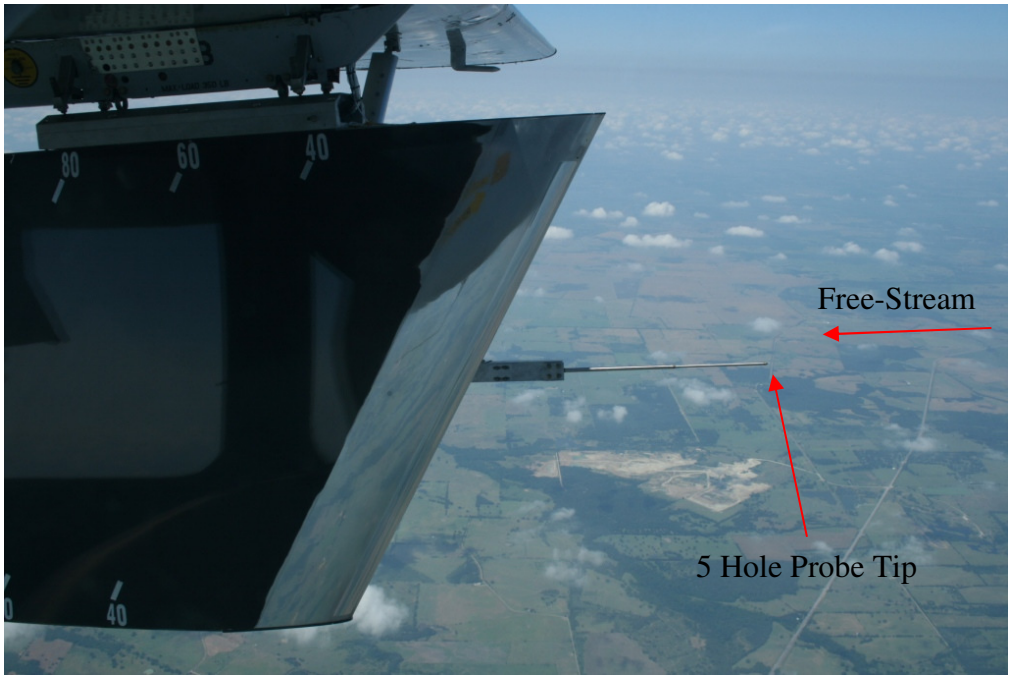


Fig. 21 Five hole probe installed on the SWIFT as seen from the cockpit.

In a CFD solution, flight conditions can only be specified at boundaries and not at local interior points, therefore the upstream flow angle must be specified such that the local flow angle at the 5-hole probe tip matches the values recorded in flight, to this end, an iterative process is used to converge on the appropriate upstream conditions. Assuming that β_{ac} has a measurement error of -1^{o} yields a good initial guess for the upstream condition. After the initial solution is obtained a correction factor based on the discrepancy between the desired and measured flow angles at the probe tip is applied to the upstream condition. Applying an under-relaxation factor of

0.8 to the angle correction and repeating this procedure typically yields convergence to within one-hundredth of a degree in as few as 3 iterations. Specifying the correct flow angle brought the CFD solution extremely close to the experimental results as shown in Fig. 22 for TP31, with Fig. 23 showing results for TP27. The FRL requested CFD simulation of these two test points based on promising flight test results with DREs applied as per preliminary low-order calculations from a separate study. In flight tests, C_P data are sampled over a five second interval, during this time-span the measurement taken from each pressure port varies slightly. The error bars in the experimental C_P curves were calculated by the FRL based on the worst case data variance observed at a given pressure port during the sampling period of the corresponding test point.

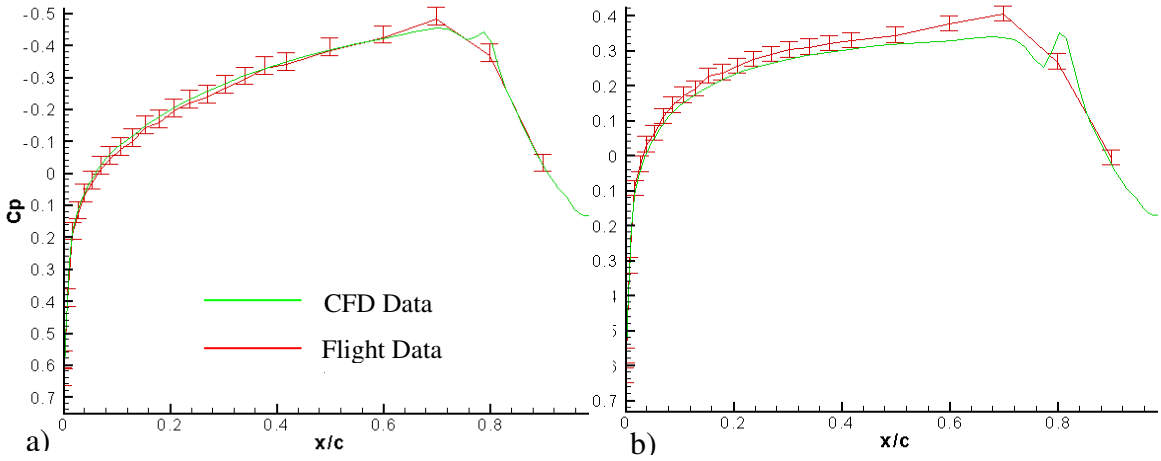


Fig. 22 C_P data showing excellent agreement between CFD and flight data for TP31. a) Inboard station. b) Outboard station.

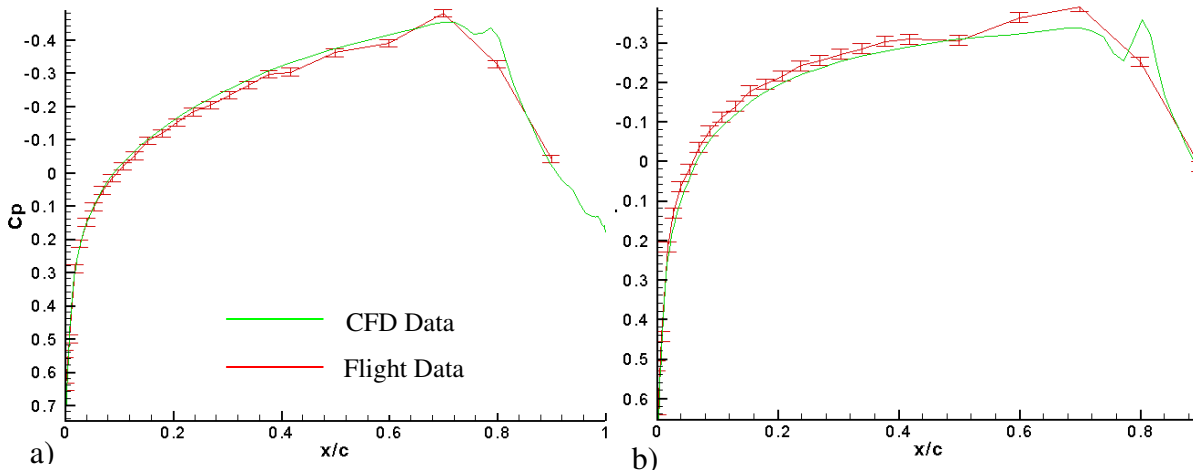


Fig. 23 C_P data showing excellent agreement between CFD and flight data for TP27. a) Inboard station. b) Outboard station.

B. Solver Options

To correctly configure the solver options, a trade study was performed to evaluate the changes incurred by each option with the correct configuration assumed to be that which most closely matches the experimental C_P data. Based on the flight regime and flow characteristics, many solver options were outright discarded. FLUENT$_{TM}$ recommends that a second order momentum discretization might be more accurate for an unstructured mesh, therefore this option will be considered. However the higher order QUICK and MUSCL schemes are only recommended for swirling or rotating flows, therefore these are not considered. The standard pressure discretization scheme is recommended for most cases, the PRESTO scheme is recommended for flows involving large amounts of curvature, the 2nd order scheme is only recommended for compressible flows, and the body-force-weighted is specific to cases where large body forces are present. Based on these

recommendations the standard and PRESTO schemes are most applicable to this study. The FLUENT$_{TM}$ manual mentions no difference in accuracy between the SIMPLE and SIMPLEC pressure-velocity coupling schemes. The only advantage offered by SIMPLEC is the enhanced under-relaxation which increases the numerical stability for skewed meshes. The PISO and FSM pressure-velocity coupling methods are recommended for unsteady flows, therefore only the SIMPLE and SIMPLEC are evaluated in this study. FLUENT$_{TM}$ offers a wide range of turbulence models from the 1-equation Spalart-Allmaras[19] model to Large Eddy Simulation (LES), however this work is not concerned with modeling turbulence therefore it is only evaluated in this trade study to determine if laminar separation is occurring aft of the minimum pressure location. For turbulence modeling the k-ε model was chosen for evaluation because it is considered the "workhorse" of turbulence models and the Spalart-Allmaras[19] model was chosen to evaluate any difference between a 2-equation model and 1-equation model. Table 8 shows the solver options to be tested; the italicized options are the FLUENT$_{TM}$ defaults.

Table 8 Solver options evaluated in this study.

Parameter	Options
Momentum Discretization	*1st Order Upwind Scheme*
	2nd Order Upwind Scheme
Pressure Discretization	*Standard*
	PRESTO!
Pressure-Velocity Coupling	*SIMPLE*
	SIMPLEC
Viscous Model	*Laminar*
	Spalart-Allmaras Turbulence Model
	k-ε Turbulence Model

Comparison of the options for momentum discretization, pressure discretization, and pressure-velocity coupling using TP27 as a test case is provided in Fig. 24. Figure 25 provides comparison of the laminar, k-ε, and Spalart-Allmaras[19] viscous models also using the flight conditions from TP27. Convergence could not be obtained using the higher order discretization schemes with the SIMPLE pressure-velocity coupling algorithm, therefore the SIMPLEC algorithm was used in conjunction with the PRESTO! and 2nd Order schemes which yielded a stable solution.

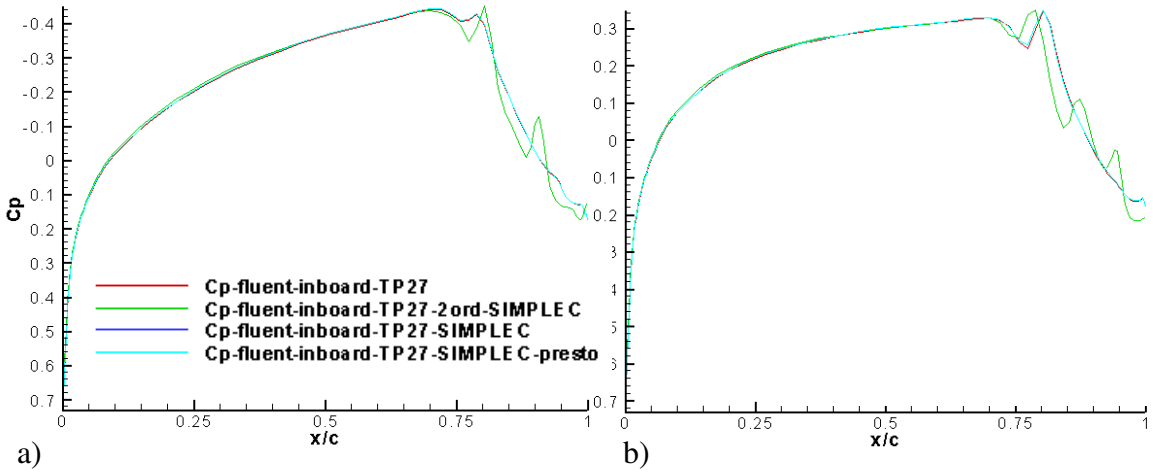


Fig. 24 C_P comparison of varying solver options. a) Inboard station. b) Outboard station.

The region in the C_P plots aft of the minimum pressure is not considered in this comparison since the neutral point location sought in this study is typically less than 5% chord and the boundary-layer stability analysis to be performed in this study will be limited to the forward portion of the wing. It is apparent from this trade study

that none of the different solver discretization or coupling schemes deviated from the baseline solution.

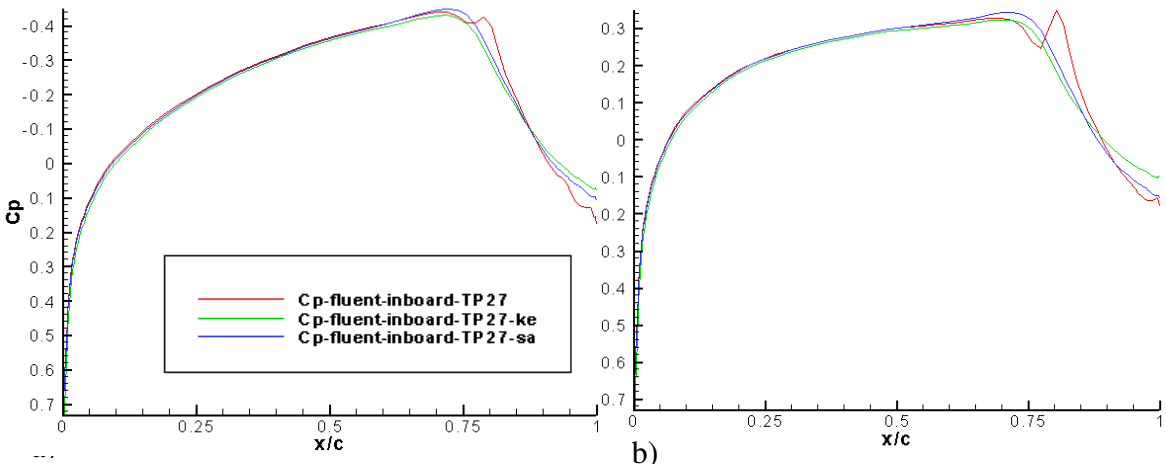


Fig. 25 C_P plots comparing different turbulence models.

The turbulence models showed very little deviation from the laminar solution with the exception of the area just aft of the minimum C_P where there appears to be a small laminar separation zone. The laminar separation bubble is small enough not to affect the overall solution and is assumed to have negligible impact on the boundary-layer. Furthermore a laminar solution must be used for the forthcoming stability analysis since instabilities in a laminar boundary-layer are responsible for transition; therefore the laminar option is retained. The results of these trade studies are that all default solver options in Table 8 will be retained for further calculations.

The basic-state solutions for TP27 and TP31 generated with FLUENT$_{TM}$ using the default solver options showed excellent agreement with C_P data taken from

flight tests and will be used for boundary-layer stability calculations detailed in Chapter V. Extraction of the boundary-layer profiles from FLUENT$_{TM}$ is the next major obstacle as the built-in post-processing tools do not provide an effective means of reconstructing grid points and their respective flow-field variables; an effective solution to this problem is also addressed in Chapter V.

CHAPTER V

STABILITY CALCULATIONS

The Langley Stability and Transition Analysis Code (LASTRAC$_{TM}$) is used in this study to perform stability analysis of the SWIFT boundary-layer profiles generated by FLUENT$_{TM}$. The focus of this stability analysis is calculation of the stabilizing and destabilizing spanwise wave-numbers and their corresponding chordwise neutral point where the disturbance amplitude first begins to grow. The placement of the DREs is such that they span the suction side of the model fixed at the chordwise neutral point. To export the boundary-layer profiles directly from FLUENT$_{TM}$ for use with LASTRAC$_{TM}$, a User Defined Function (UDF) must be written to pull the flow-field variables at specific spanwise planes and reconstruct the computational grid.

A. Grid Reconstruction

A UDF is a code written in the C programming language which is dynamically loaded and compiled in the FLUENT$_{TM}$ environment allowing the user to perform calculations beyond the standard abilities of the flow solver. Examples of common UDFs include defining custom boundary conditions, material properties, or extending the post-processing abilities of FLUENT; these UDFs are typically small codes and are limited in scope. The most important step in the use of this function is to create a well-constructed grid around the wing to be analyzed while generating the basic-state mesh in GAMBIT$_{TM}$. The UDF is customized to reconstruct the

structured portion of the hybrid mesh developed in Chapter III. The initial grid must be constructed carefully as highly skewed cells, always undesirable in CFD, will likely cause the UDF to crash while attempting to reconstruct the grid.

The UDF first extracts the flow field data at each grid point from a pre-defined surface. FLUENT$_{TM}$ exports the grid point data in no specific order, thus the first task of this function is to accurately reconstruct the grid points into a coherent data structure. The UDF first establishes the surface points by their satisfaction of the no-slip condition and orders both the upper and lower surface points beginning with the stagnation point and moving downstream as shown in Fig. 26. Building off of the surface points, the function identifies the column of points related to each surface point based on proximity and dot-product. For reconstruction of the first row of points away from the surface, the function compares the dot-product of the vector between two points with the surface normal. In this way the algorithm "looks" for the column in a region near the surface normal preventing misidentification of neighboring nodes. After the first row of points away from the surface is established the algorithm calculates the vector connecting the first two nodes in each column and uses this for the dot-product comparison instead of the normal vector as the grid is not expected to be orthogonal. With both streamwise and normal indices now in place, it is possible to construct an ordered data set.

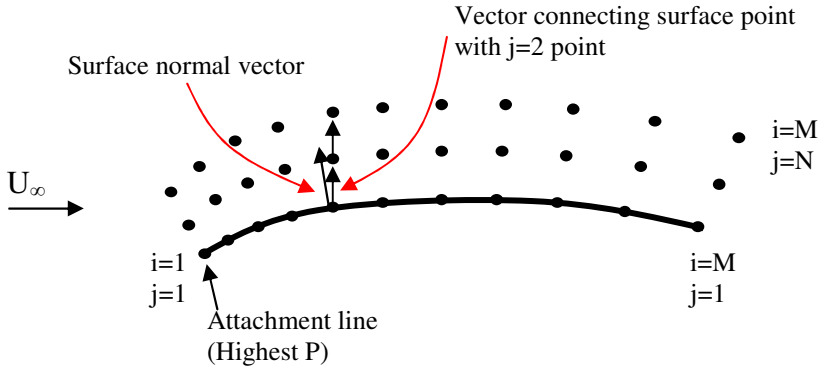


Fig. 26 Sketch of grid reconstruction algorithm employed by the UDF.

With the reconstructed basic-state data now in hand, the next task is to construct an appropriate stability grid which must be orthogonal. Since the grid created with GAMBIT$_{TM}$ will very likely not be perfectly orthogonal, the code is forced to construct its own orthogonal grid by shooting out normal lines from the surface. Variables at the intersection of the normal lines and the existing circumferential lines must be interpolated; this task is performed using a cubic spline fit and is illustrated in Fig. 27.

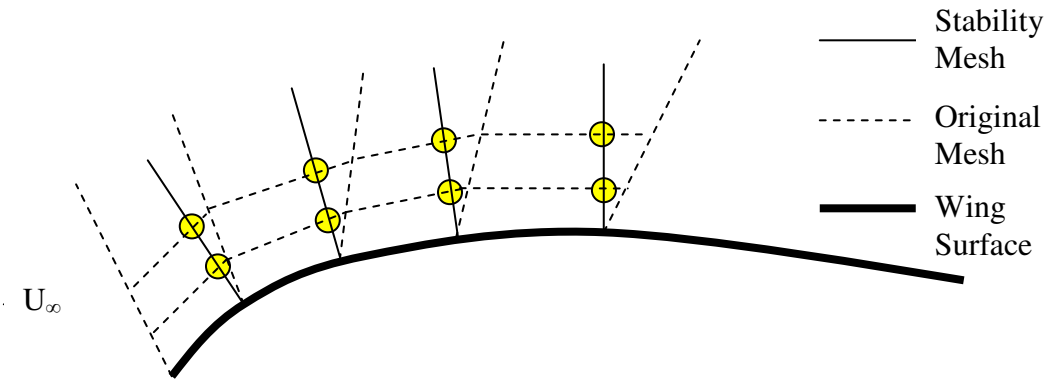


Fig. 27 Sketch of the circumferential spline fit process.

At this point all that remains to be done by the UDF is the clustering of grid points normal to the surface. Stability analysis requires a large number of points near the surface in order to resolve the steep disturbance gradient at the wall and to capture the critical layer within the boundary-layer; therefore a cubic spline interpolation scheme is employed in the direction normal to the wall allowing the user the freedom to increase the number of points within the boundary-layer. LASTRAC$_{TM}$ performs its own basic-state normal spline fit therefore this feature is not needed for stability calculations but is necessary for further post-processing of the boundary-layer profiles. Therefore the UDF writes both an interpolated basic-state database with wall normal clustering, and a non-interpolated database. Clustering of the points normal to the surface is handled with a cosine function given in equation (1).

$$\begin{aligned} \zeta_i &= \tau \cos(\theta_i) \\ \theta_i &= \theta_{i-1} + \Delta\theta \\ \Delta\theta &= \frac{\frac{\Pi}{2}}{N} \end{aligned} \tag{1}$$

The original normal coordinate y is mapped to ζ based on the number of points "N" to be fit in the stability grid. The symbol τ is the product of η, the similarity boundary-layer length scale from the Blasius solution, and σ, a scaling constant specified by the user such that the stability grid captures the outer edge of the boundary-layer, this quantity is shown in equation (2).

$$\tau = \sigma\eta = \sigma\sqrt{\frac{\nu x}{u_e}} \tag{2}$$

The source code of the grid reconstruction and interpolation UDF is provided in Appendix A, Figures A – 1 through A – 4 show the output from this code as the evolution of the grid from the random data structure as output by FLUENT$_{TM}$ to the boundary-layer fitted, orthogonal mesh. With the stability grid now constructed, the necessary boundary-layer profiles can be passed to the stability solver.

B. Boundary-layer Stability Calculations

LASTRAC$_{TM}$ is a robust stability analysis tool applicable to a wide range of basic-state flow types. For this study, LASTRAC$_{TM}$ is configured for analysis of an infinite-span swept-wing boundary-layer with a body-fitted coordinate system as shown in Fig. 28 where x is tangent to the surface and perpendicular to the leading edge, z is tangent to the surface along the leading edge, and y is perpendicular to the surface. For the infinite-span swept-wing all derivatives in the spanwise direction are assumed to be zero.

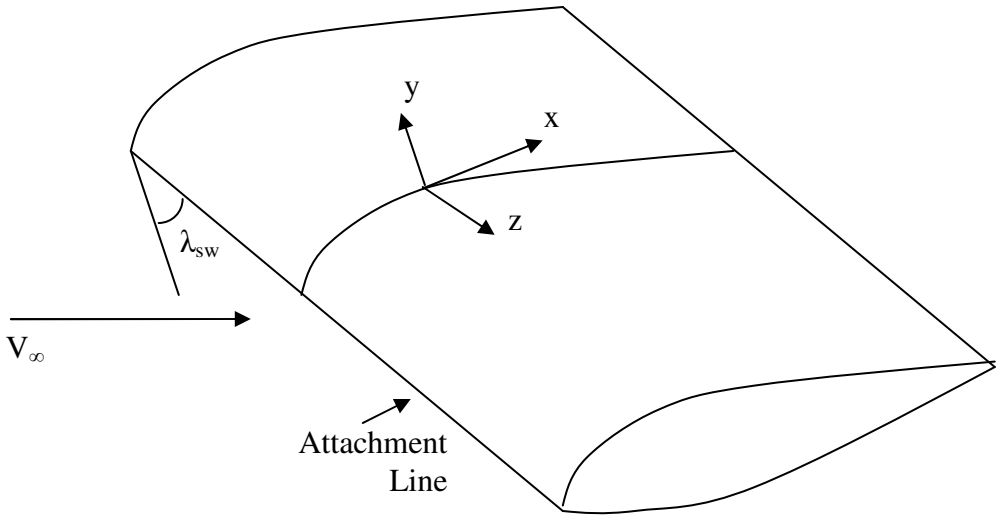


Fig. 28 Body fitted infinite-span swept wing coordinate system used by LASTRAC$_{TM}$.

The current build of LASTRAC$_{TM}$ can be used in three modes: Linear (Parallel) Stability Theory (LST), Linear Parabolized Stability Equation (PSE), and Non-linear Parabolized Stability Equation (NPSE). For all formulations the disturbance equations are derived from the Navier-Stokes equations by substituting flow-field variables as the superposition of a basic-state and disturbance quantity as demonstrated in Eq. 3 where φ represents any of the flow-field variables.

$$\varphi(x, y, z, t) = \Phi(x, y) + \varphi'(x, y, z, t) \tag{3}$$

After the above form is substituted for each variable, terms consisting of purely basic-state quantities automatically satisfy the original conservation equations and thus can be eliminated. Non-linear terms appear as those containing the product of two or more disturbance quantities. The LST and PSE being linear formulations,

discard these non-linear terms since disturbances are assumed to be very small compared with the basic-state while the NPSE retains the bulk of the non-linear terms. After some manipulation one can organize the system into the classical disturbance equations with all non-linear terms are on the right hand side allowing the linear formulations to be obtained by setting the left hand side equal to zero.

C. Linear Stability Theory

The derivation of the LST equations begins with the previously mentioned disturbance equations. The basic state terms are assumed to be parallel in that they are functions of the y-direction only, therefore: V≡0, ρ= ρ(y), U=U(y), W=W(y), and T=T(y). The next step in deriving the LST equations is to perform a normal mode analysis. Through use of a Fourier-Laplace transformation, it can be shown that disturbances assume the following form:

$$\varphi'(x, y, z, t) = \tilde{\varphi}(y) e^{i(\alpha x + \beta z - \omega t)} + c.c. \tag{4}$$

In Eq. 4 φ´ is the disturbance vector consisting of (ρ´, u´, v´, w´, T´), $\tilde{\varphi}$ is the disturbance magnitude vector consisting of ($\tilde{\rho}$, $\tilde{u}$, $\tilde{v}$, $\tilde{w}$, $\tilde{T}$), and "c.c." stands for complex conjugate. LASTRAC$_{TM}$ considers the spatial stability problem where ω, the non-dimensional frequency shown in Eq. 5, is a real quantity while α and β, the respective streamwise and spanwise wave-numbers, are complex.

$$\omega = \frac{2\pi\eta}{u_e} f \tag{5}$$

In spatial stability, the imaginary components of α and β represent the disturbance growth rates in their respective directions with a negative sign indicating

instability. Since the infinite-span swept-wing assumption set all spanwise derivatives to zero, the spanwise growth rate (β_i) can also be assumed to be zero. The infinite-span swept-wing assumption is justified based on Fig. 29 which shows iso-lines of pressure on the test side of the SWIFT taken from the CFD solution. In the region just aft of the attachment line up to ~10% chord where neutral point calculations will be conducted, the iso-lines are nearly aligned with the leading edge behaving similar to an infinite-span swept-wing.

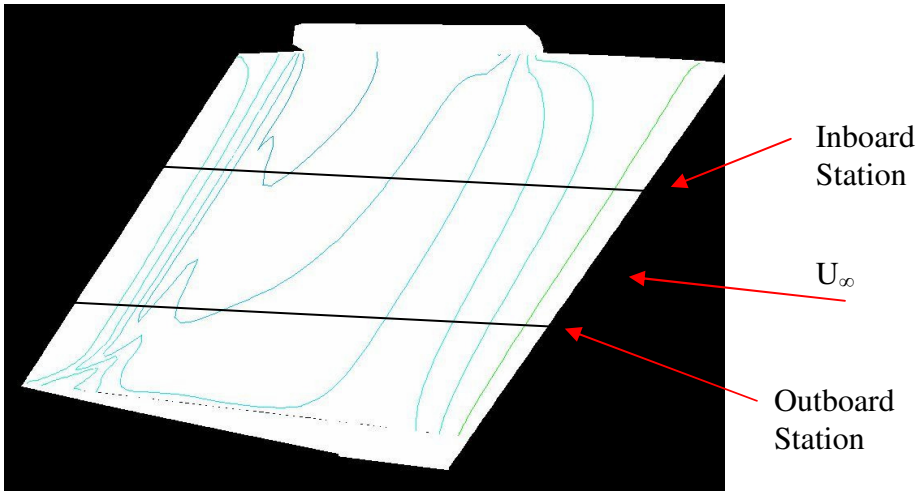


Fig. 29 Iso-lines of pressure verifying the infinite-span swept-wing assumption.

Since the SWIFT is designed with a favorable pressure gradient, the Tollmein-Schlichting (T-S) instability is subcritical, therefore two-dimensional (2-D) disturbances have negligible effect on transition. Because flight testing is a low disturbance environment, the development of traveling crossflow vortices is not expected. Furthermore the DREs eliminate any traveling crossflow vortices by

enforcing a stationary spanwise mode. Because the crossflow will not be time dependent the non-dimensional frequency ω is set to zero. The real components of α and β are the wave numbers in their respective directions and are given in Eq. 6 where λ_x and λ_z are the streamwise and spanwise non-dimensional wavelengths.

$$\alpha_r = \frac{2\pi}{\lambda_x}, \beta_r = \frac{2\pi}{\lambda_z} \tag{6}$$

At this point the LST equations take the form of a general complex eigenvalue problem for the complex streamwise wavenumber α (eigenvalue), and the complex disturbance magnitude vector $\tilde{\varphi}$ (eigenvector) where a range of β_r is specified by the user. The periodic spanwise spacing of the DREs corresponds to the spanwise wavelength (λ_z).

In this study LST is used to predict the Branch I neutral point which was shown by Radeztsky et al.[7] to be the most effective chord-wise placement of the DREs. The disturbance growth in a typical laminar boundary-layer is shown in Fig. 30 where disturbance amplitude is plotted vs. R, the square-root of the x Reynolds number. The general trend shows that for a short distance downstream of the attachment line, disturbances decay until the Branch I neutral point where they begin to grow until peaking at the Branch II neutral point.

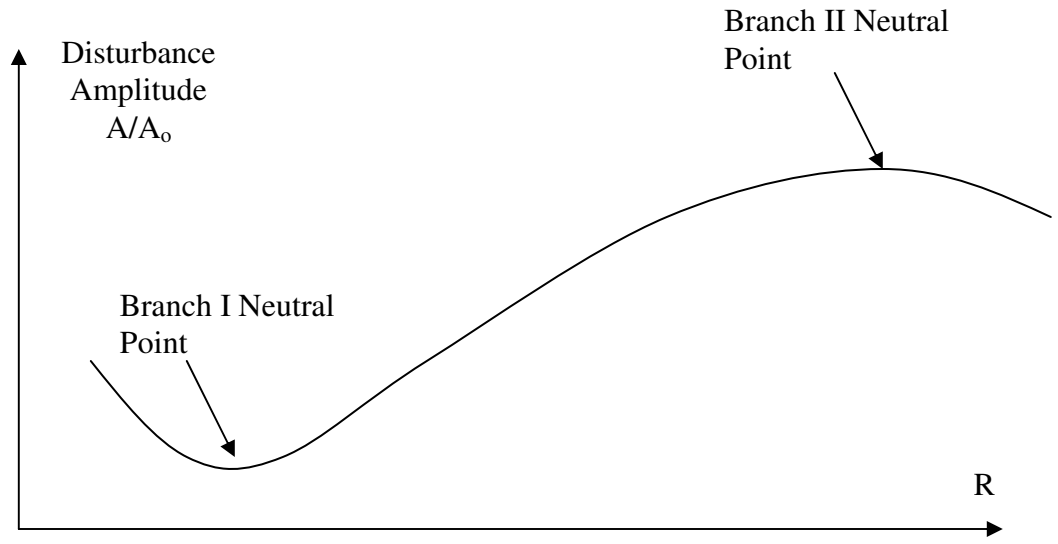


Fig. 30 Sketch of disturbance growth in a typical boundary-layer.

Since disturbances decay in the region upstream of the neutral point so too will the crossflow modes influenced by the roughness elements thus lowering their effectiveness. Similarly, Radeztsky et al.[7] observed that placing roughness elements too far downstream of the neutral point also minimized their influence.

In the derivation of the LST, non-linear terms consisting of the product of two or more disturbance quantities were discarded since the disturbances are considered small. This assumption is most valid near the neutral point where disturbances are at their smallest making the LST sufficiently accurate for calculation of the neutral point. The crossflow instability has been shown to exhibit strong non-linear effects (Reibert et al.[6]), however the non-linear effects develop

downstream of the neutral point rendering LST an accurate tool for prediction of the neutral point of a crossflow dominated flow.

Using LASTRAC$_{TM}$ to compute LST analysis with the mean-flow generated by FLUENT$_{TM}$ yields the disturbance growth shown in Fig. 31 for the inboard and outboard stations at the flight conditions corresponding to TP27 and TP31.

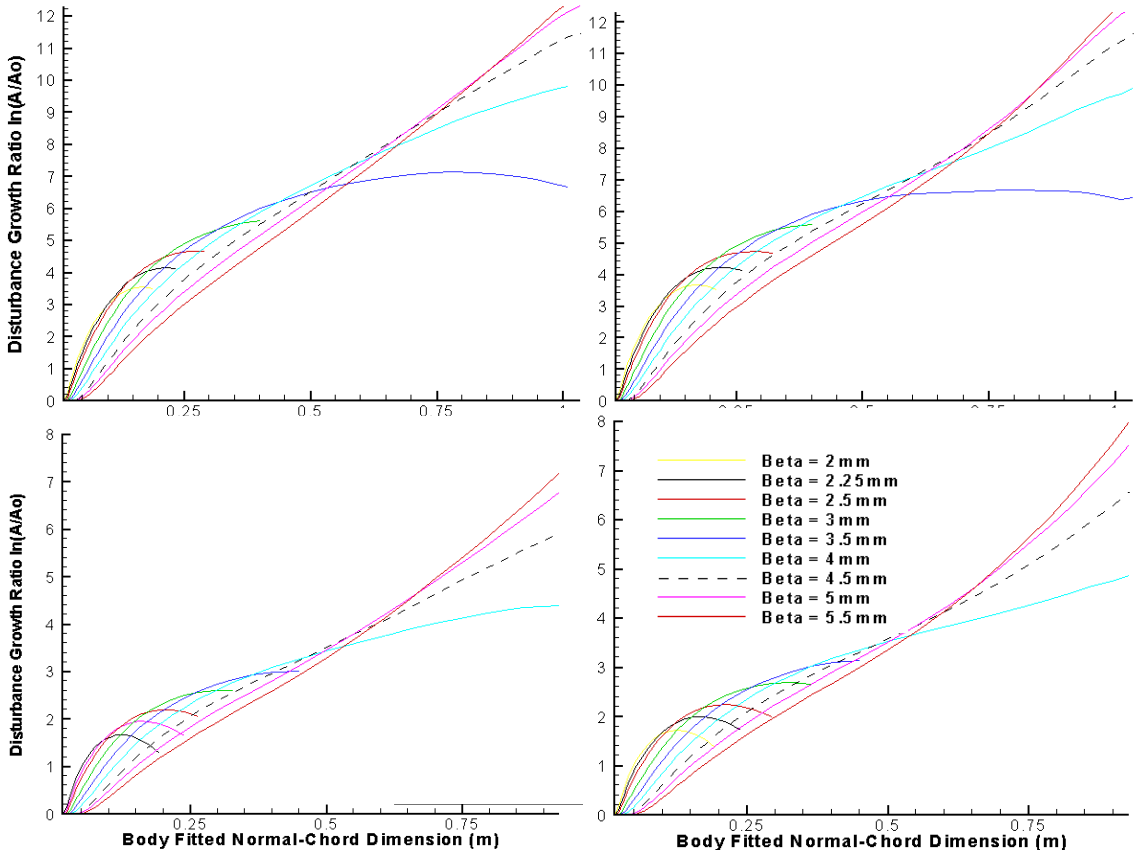


Fig. 31 LST Disturbance growth calculated with LASTRAC$_{TM}$. a) TP27 Inboard. b) TP27 Outboard. c) TP31 Inboard. d) TP31 Outboard.

Since LASTRAC$_{TM}$ interpolates its own grid based on the basic-state data, the user is free to specify a larger number of points in the boundary-layer than that

which is provided in the boundary-layer database, however care must be taken to not specify more points than can accurately be interpolated from the original boundary-layer profiles. Similarly the user can set the stability grid truncation location to be outside the grid provided in the database, this is often necessary to satisfy the free-stream disturbance boundary condition. In order to show convergence of the interpolated stability mesh, both the number of points in the grid and the truncation location were increased until the neutral point and disturbance growth ceased to change. Table 9 provides the LST solver and grid parameters specified in LASTRAC$_{TM}$.

Table 9 Parameters specified in LASTRAC$_{TM}$ calculations.

Option	Value
grid_type	wall_cluster
num_normal_pts	101
num_normal_pts_geig	101
ymax	30
ymax_glog_search	30
marching_method_2d	along_station
init_station	10
final_station	60
solution_type	local_eig_solution
freq_unit	in_hertz_freq
beta_unit	in_mm_beta
freq	0*10
beta in mm	2, 2.25, 2.5, 3, 3.5, 4, 4.5, 5, 5.5, 6
qp_approx	True

The method for spanwise spacing of DREs is outlined by Reibert et al.[6], with the main idea being to force modes whose disturbances first grow and then decay

before causing transition. These forced modes distort the mean-flow enough to inhibit the naturally occurring crossflow vortices which serve as the primary instability in a swept-wing boundary-layer. Thus transition is delayed. First and foremost delay of transition and achievement of laminar flow is sought, but the DREs should also be able to excite the unstable modes thus forcing transition and achieving full chord turbulent flow. To achieve both goals, the DRE spanwise spacing of 2.25mm is chosen for the SWIFT such that actuating all elements excite a mode which will delay transition while actuation of every second element will excite the extremely unstable 4.5mm mode thus tripping the flow and forcing turbulence. These spanwise modes correspond to those calculated in earlier stability calculations based on these flight conditions but with boundary-layer profiles generated using a boundary-layer code assuming an infinite-span swept-wing. The neutral point for the very unstable (4.5mm) mode is used for the chordwise placement of the DRE array whether achieving LFC with wavelength of 2.25mm or enhancing turbulence with the 4.5mm spacing. The neutral points for both test points corresponding to the 4.5mm spacing is shown in Table 10. The chordwise placement of the DREs array for each test point will be along a line connecting these two points as sketched in Fig. 32.

Table 10 Neutral point calculations for TP27 and TP31.

Test Point	Inboard Station Neutral Pt.	Outboard Station Neutral Pt.
27	1.65% Chord	1.45% Chord
31	2.07% Chord	2.07% Chord

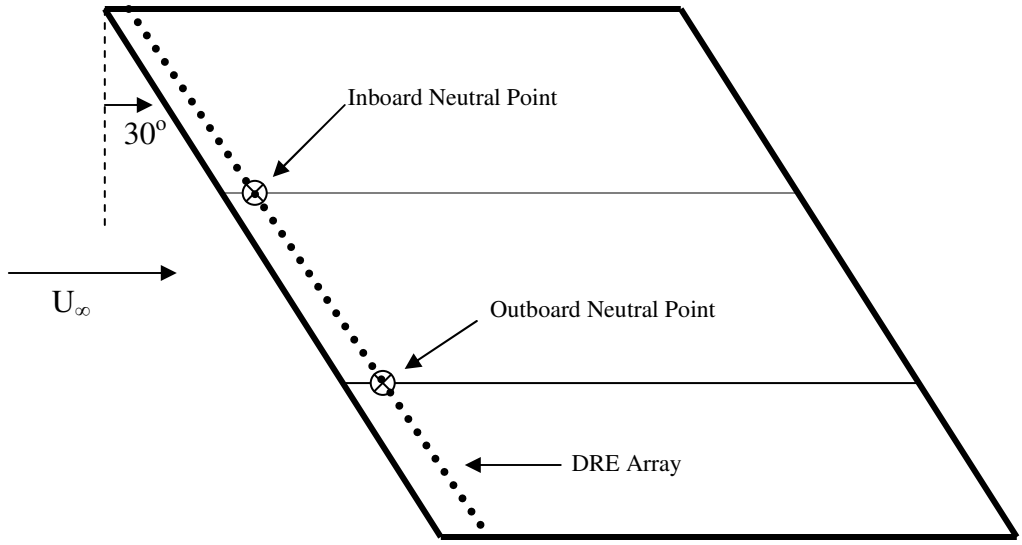
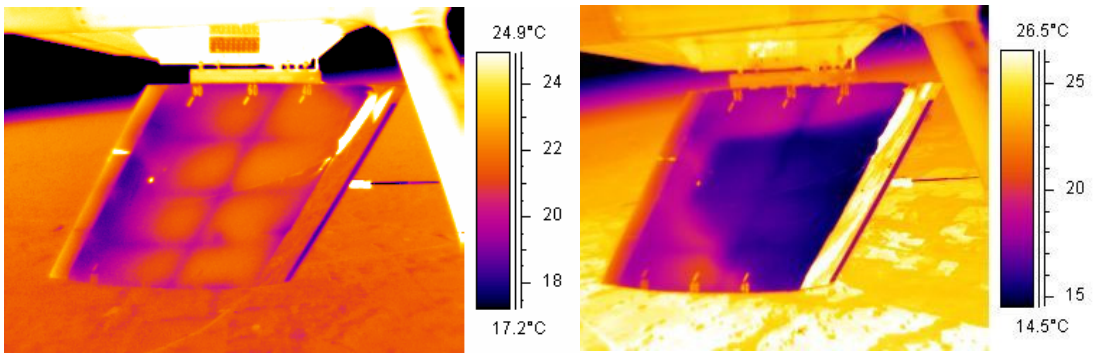


Fig. 32 Sketch of DRE placement based on neutral point calculations.

Preliminary flight tests conducted by the FRL at TP31 (-2.61° α_{SW}) have shown the 4.5mm spaced DREs to effectively excite instability at a roughness height of only 30μm when placed at 2.07% chord inboard and 2.41% chord outboard. The chordwise location at the outboard station was calculated in a previous computational study however the location at the inboard station was based on observations from other flight tests. One can see that this DRE configuration achieves full-span turbulent flow as shown through IR thermography in Fig. 33. For TP31 the neutral point calculations in this study have been shown in flight to be extremely accurate for the inboard station; however flight tests have yet to be performed with 30μm roughness applied to the outboard DRE placement of 2.07% chord as predicted in this study. Since the 2.41% chord outboard neutral point proved

effective in flight, this placement will be retained by the FRL for further tests at TP31. The DRE placement predicted for TP27 has yet to be investigated in flight since current resources are directed towards further study of TP31.



a) b)

Fig. 33 IR images taken in flight showing the DREs exiting the unstable 4.5mm mode.[†] a) Turbulent flow with the DREs applied. B) Mostly laminar flow without the DREs applied.

The LST calculations detailed in this section have been used to determine the selection of spanwise DRE spacing and chordwise location. The neutral points calculated for the 4.5mm wave-number are used by the FRL for chordwise placement of the DRE array on the SWIFT leading edge to excite instability. First a temporary array will be attached consisting of removable adhesive "dots", these will be used to verify DRE placement in flight. Pending flight test verification a pneumatic actuator DRE system will be installed in the leading edge of the SWIFT to achieve active Laminar Flow Control in flight tests.

[†] IR photographs courtesy of Andrew Carpenter of the FRL

CHAPTER VI

SUMMARY AND CONCLUSIONS

More than ever in commercial aviation, emphasis is being placed on increasing efficiency and reducing operating cost. Laminar Flow Control (LFC) is a promising means to immediately reduce the drag of subsonic transport aircraft and thus reduce the amount of fuel consumed. This work is a computational study coordinated with flight research concerned with using 3-D roughness elements (DREs) as a means of maintaining laminar flow over a swept wing. This chapter reviews the important topics investigated in this work and summarizes the pertinent results.

1. A Hybrid Meshing technique was developed which allowed construction of a viscous computational mesh of the O-2 and SWIFT while in flight. Trade studies were performed to validate the chosen geometric configuration of the mesh. A solver trade study showed the default solver options and laminar viscous model to be the best configuration. The CFD solution was used to validate placement of a free-stream data probe allowing more accurate measurement of flight conditions. The end result was a CFD basic-state solution that showed excellent agreement with flight C_P measurements.

2. A FLUENT$_{TM}$ UDF was developed that allows the user to extract boundary-layer data from a 3-D solution and reconstruct a 2-D "slice" of the computational grid. The boundary-layer profiles are interpolated at grid points which are orthogonal to the surface, constructing a suitable mesh for stability analysis of the profiles. The boundary-layer data can then be compiled into a mean-flow file format suitable for a specific stability analysis code.

3. Linear Stability analysis was performed using LASTRAC$_{TM}$ on the basic-state boundary-layer profiles exported from FLUENT$_{TM}$. The disturbance growth for varying spanwise wave numbers was used to identify an extremely unstable 4.5mm mode as well as identifying the 2.5mm as a good candidate for stabilizing the 4.5mm mode for both flight conditions of interest TP27 and TP31. The neutral points of the 4.5mm mode at TP27 were calculated to be 1.65% chord for the inboard and 1.45% chord for the outboard station while at TP31 the neutral points were found to be 2.07% chord for both stations.

4. Preliminary flight tests have shown that for TP31 the 2.07% chord-wise placement of the 4.5mm DREs effectively excited instability with 30μm roughness height. The spanwise spacing and chord-wise placement determined from this study will be used for further flight test investigations

into the effectiveness of the DREs for both suppressing and exciting the crossflow instability of the SWIFT model.

5. Close work between this computational study and the experimental work done by the FRL have enhanced both efforts greatly. The FRL has been an invaluable source of knowledge concerning the flight configuration and test equipment. Furthermore interaction with the experimental researchers has broadened the perspective of the author and enriched the computational research. Without the flight C_P data there would be no means of validating the CFD analysis. Without the ability of the CFD analysis to reconstruct the flow-field in regions where placement of data probes is not feasible, the experimental research would not be able to account for all possible sources of error. Each effort has enhanced the other such that the combined effect is greater than the sum of the two parts.

REFERENCES

[1] Lachmann, G.V., (Ed) *Boundary-layer and Flow Control*, Vol. 2, Pergamon London, 1961.

[2] McCartney S., "Flying Stinks – Especially for the Airlines", *Wall Street Journal*, June 10 2008, Page D1.

[3] Sturgeon, R.F., "The Development and Evaluation of Advanced Technology Laminar-Flow-Control Subsonic Transport Aircraft." AIAA Paper No. 78-96, 1978.

[4] Green, J.E., "Laminar Flow Control – Back to the Future?" AIAA Paper No. 2008-3738, 2008.

[5] Saric W.S., Reed H.L., and White E.B., "Stability and Transition of Three-Dimensional Boundary-layers". *Annu. Rev. Fluid Mechanics*, Vol. 35, 2003, pp. 413-40.

[6] Reibert, M.S., and Saric, W.S., " Review of Swept-wing Transition." AIAA Paper No. 97-1816, 1997.

[7] Radeztsky, R.H., Reibert, M.S., and Saric, W.S., "Effect of Micron-Sized Roughness on Transition in Swept-Wing Flows." AIAA Paper No. 93-0076, 1993.

[8] Saric, W.S., Carrillo, R.B., and Reibert, M.S., " Leading-Edge Roughness as a Transition Control Mechanism." AIAA Paper No. 98-0781, 1998.

[9] *GAMBIT$_{TM}$, Software Package*, Ver 6.3.16, ANSYS Inc., Canonsburg PA, 2008.

[10] *FLUENT$_{TM}$ Software Package*, Ver 6.3.26, ANSYS Inc., Canonsburg PA, 2008.

[11] *LASTRAC$_{TM}$ Software Package*, Ver 1.2, NASA Langley Research Center, Hampton VA, 2004.

[12] Saric W.S., Carpenter A.L., Hunt L.E., and Kluzek C.D. "SWIFT – Flight Test Plan for Swept-Wing Experiments." TAMUS-AE-TR-06-001, Technical Report, January 2006.

[13] Saric W.S., Carpenter A.L., Hunt L.E., and Kluzek C.D., "Cessna O-2 General Flight Test Procedures: Operations for the Flight Research Laboratory." TAMUS-AE-TR-06-003, Technical Report, January 2006.

[14] Saric W.S., Carpenter A.L., Hunt L.E., McKnight C.W., and Schouten S.M.. "SWIFT – Safety Analysis for Swept-Wing Experiments." TAMUS-AE-TR-06-002, Technical Report, January 2006.

[15] McKnight, C.W. " Design and Safety Analysis of an In-Flight, Test Airfoil." Masters Thesis, Texas A&M University, College Station, 2006.

[16] Zuccher S., and Saric W.S., "Infrared Thermography Investigations in Transitional Supersonic Boundary-layers." Exps. in Fluids, Vol. 44, 2008, pp. 145-157.

[17] Saric W.S., Reed H.L., and Banks D.W., " Flight Testing of Laminar Flow Control in High-Speed Boundary-layers." RTO-MP-AVT-111/RSM, 2005.

[18] *Solidworks$_{TM}$ Educational Edition Software Package*, Solidworks Corporation, Concord, Massachusetts, 2007.

[19] Spalart, P.R., and Allmaras, S.R., "A One-Equation Turbulence Model for Aerodynamic Flows." AIAA Paper No. 92-0439, 1992.

APPENDIX A

This section contains the source code used to extract and reconstruct boundary-layer profiles from FLUENT$_{TM}$.

```
/* orders the points output from fluent */

#include <udf.h>

#include "spline_h.h"
#include "theta_h.h"

#define cutoff 10.
#define c_cut 0.95
#define norm_st 30
#define Pi 3.14159
#define sw_ang 31.3


DEFINE_ADJUST(print_f_centroids, domain){

        int n_pts, i, j, k, n, n_st, n_norm, sum, tei, o, l, k_end;
        int pto, ptn, stag, cond, n_upper, n_lower, pt, stab_j=40, LE;
        int *b, *pti, *ne1, *ne2, *cut;
        double x_vec[ND_ND], y_vec[ND_ND], z_vec[ND_ND];
        double sqrt(double), fabs(double), atan(double);
        double cos(double), sin(double), theta_find(double, double);
        double dum1, dum2, dum3, dum4, del_old, dum1_old, chord, del, scale = 2.0;
        double nx_dn1, ny_dn1, nx_up1, ny_up1, x_dum, y_dum, u_dum, v_dum, dot = 0.9;
        double z_dum, w_dum, ypp1, yppn, s_p = 0., stag_x, Rc;
        double v1x, v1y, v2x, v2y, v3x, v3y, v4x, v4y, mn, mg, bob;
        double v_tot, v_tot_old, theta, d_theta, ang;
        double th1, th2, th3, th4, th_norm;
        double *x, *y, *s, *ypp, *s_th;
        double *u, *v, *w, *p, *rho;
        double *x_int, *y_int, *u_int, *v_int, *w_int, *p_int, *rho_int;
        double *nx_up, *ny_up, *nx_dn, *ny_dn;
        double **y_up, **x_up, **u_up, **v_up, **w_up, **rho_up, **p_up;
        double **y_dn, **x_dn, **u_dn, **v_dn, **w_dn, **rho_dn, **p_dn;
        double **x_int2, **y_int2, **x_sp, **y_sp, **u_sp, **v_sp, **w_sp;
        double **p_sp, **rho_sp;

        /* UDF vars */
        double FC0[ND_ND], FC1[ND_ND], FF[ND_ND], A[ND_ND];
        double A1=0., A0=0.;
        double x1=0., x0=0., z1=0., z0=0., y1=0., y0=0.;
        double u0=0., u1=0., v0=0., v1=0., w0=0., w1=0.;
        double sqrt(double);
        cell_t c0, c1;
        face_t f, f_indx;
        int ID = 36, m=1;
        Thread *th = Lookup_Thread(domain, ID), *tc0, *tc1, *tf0, *tf1;

        /* end UDF vars */

        FILE *ofp, *ofp2, *ofp3, *ofp4, *ofp5, *ofp6, *ofp7, *ofp8;
        FILE *ofp9;

        ofp = fopen("ordered_up_norm.txt", "w");
        ofp2 = fopen("ordered_lower.txt", "w");
        ofp3 = fopen("ordered_up_las-sc.txt", "w");
```

```
        ofp4 = fopen("ordered_up_las-xc.txt", "w");
        ofp5 = fopen("spline_bl_las-xc.txt", "w");
        ofp6 = fopen("ordered_up_norm-sp.txt", "w");
        ofp7 = fopen("spline_debug.txt", "w");
        ofp8 = fopen("root_debug.txt", "w");
        ofp9 = fopen("ordered_up_pre.txt", "w");
        /* assign direction vectors based on coordinate system*/


         x_vec[0] = 0;
         x_vec[1] = 0;
         x_vec[2] = -1.0;
         y_vec[0] = -1.0;
         y_vec[1] = 0;
         y_vec[2] = 0;
         z_vec[0] = 0;
         z_vec[1] = 1.0;
         z_vec[2] = 0;

        n_pts = 0;
        printf("%d\n",n_pts);
/* loop over all faces in the domain to count index for allocation */
        begin_f_loop(f, th){
                n_pts +=1;
        }
        end_f_loop(f, th)

        printf("%d\n",n_pts);

        x = (double *)malloc((n_pts + 1) * sizeof(double));
                if(x == NULL){
                    (void)fprintf(stderr, "ERROR: Malloc failed x");
                    (void)exit(EXIT_FAILURE);    /* or return EXIT_FAILURE*/
                      }
        y = (double *)malloc((n_pts + 1) * sizeof(double));
                if(y == NULL){
                    (void)fprintf(stderr, "ERROR: Malloc failed y");
                    (void)exit(EXIT_FAILURE);    /* or return EXIT_FAILURE*/
                        }
        u = (double *)malloc((n_pts + 1) * sizeof(double));
                if(u == NULL){
                    (void)fprintf(stderr, "ERROR: Malloc failed u");
                    (void)exit(EXIT_FAILURE);    /* or return EXIT_FAILURE*/
                       }
        v = (double *)malloc((n_pts + 1) * sizeof(double));
                if(v == NULL){
                    (void)fprintf(stderr, "ERROR: Malloc failed v");
                    (void)exit(EXIT_FAILURE);    /* or return EXIT_FAILURE*/
                        }
        w = (double *)malloc((n_pts + 1) * sizeof(double));
                if(w == NULL){
                    (void)fprintf(stderr, "ERROR: Malloc failed w");
                    (void)exit(EXIT_FAILURE);    /* or return EXIT_FAILURE*/
                        }
        p = (double *)malloc((n_pts + 1) * sizeof(double));
                if(p == NULL){
                    (void)fprintf(stderr, "ERROR: Malloc failed p");
                    (void)exit(EXIT_FAILURE);    /* or return EXIT_FAILURE*/
                       }
        rho = (double *)malloc((n_pts + 1) * sizeof(double));
                if(rho == NULL){
                    (void)fprintf(stderr, "ERROR: Malloc failed rho");
                    (void)exit(EXIT_FAILURE);    /* or return EXIT_FAILURE*/
                        }
        b = (int *)malloc((n_pts + 1) * sizeof(int));
                if(b == NULL){
                    (void)fprintf(stderr, "ERROR: Malloc failed b");
```

```
                    (void)exit(EXIT_FAILURE);    /* or return EXIT_FAILURE; */
                          }
        sum = 0;
        i = 1;

/* loop over all faces in the thread */
        begin_f_loop(f, th){

                c0 = F_C0(f,th);  /*returns cell index*/
                tc0 = THREAD_T0(th);
                c1 = F_C1(f,th);      /*returns cell index*/
                tc1 = THREAD_T1(th);

                C_CENTROID(FC0,c0,tc0);
                C_CENTROID(FC1,c1,tc1);

                c_face_loop(c0, tc0, n)
                {
                        tf0 = C_FACE_THREAD(c0, tc0, n);
                        if (THREAD_TYPE(tf0)==THREAD_F_WALL){
                                f_indx = C_FACE(c0, tc0, n);
                                F_CENTROID(FF, f_indx, tf0);
                                F_AREA(A,f_indx,tf0);
                                A0 = sqrt(A[0]*A[0] + A[1]*A[1] + A[2]*A[2]);
                                x0 = FF[0]*x_vec[0] + FF[1]*x_vec[1] + FF[2]*x_vec[2];
                                y0 = FF[0]*y_vec[0] + FF[1]*y_vec[1] + FF[2]*y_vec[2];
                                /*z0 = FF[0]*z_vec[0] + FF[1]*z_vec[1] +
FF[2]*z_vec[2];*/
                        }
                }
                c_face_loop(c1, tc1, n){
                        tf1 = C_FACE_THREAD(c1, tc1, n);
                        if (THREAD_TYPE(tf1)==THREAD_F_WALL){
                                f_indx = C_FACE(c1, tc1, n);
                                F_CENTROID(FF, f_indx, tf1);
                                F_AREA(A, f_indx, tf1);
                                A1 = sqrt(A[0]*A[0] + A[1]*A[1] + A[2]*A[2]);
                                x1 = FF[0]*x_vec[0] + FF[1]*x_vec[1] + FF[2]*x_vec[2];
                                y1 = FF[0]*y_vec[0] + FF[1]*y_vec[1] + FF[2]*y_vec[2];
                                /*z1 = FF[0]*z_vec[0] + FF[1]*z_vec[1] +
FF[2]*z_vec[2];*/
                                x[i] = (x0*A0 + x1*A1) / (A0 + A1);
                                y[i] = (y0*A0 + y1*A1) / (A0 + A1);
                                u[i] = 0.;
                                v[i] = 0.;
                                w[i] = 0.;
                                p[i] = F_P(f_indx,tf1);
                                rho[i] = 0.96095;
                                b[i] = 1;
                                goto Finish;
                        }
                }
                x0 = 0.;
                x1 = 0.;
                y0 = 0.;
                y1 = 0.;
                z0 = 0.;
                z1 = 0.;

                for(k=0; k<3; k++){
                x0 += FC0[k]*x_vec[k];
                y0 += FC0[k]*y_vec[k];
                /*z0 += FC0[k]*z_vec[k];*/
                x1 += FC1[k]*x_vec[k];
                y1 += FC1[k]*y_vec[k];
                /*z1 += FC1[k]*z_vec[k];*/
                }
```

```
                u0 = C_U(c0,tc0)*x_vec[0] + C_V(c0,tc0)*x_vec[1] +
C_W(c0,tc0)*x_vec[2];
                v0 = C_U(c0,tc0)*y_vec[0] + C_V(c0,tc0)*y_vec[1] +
C_W(c0,tc0)*y_vec[2];
                w0 = C_U(c0,tc0)*z_vec[0] + C_V(c0,tc0)*z_vec[1] +
C_W(c0,tc0)*z_vec[2];
                u1 = C_U(c1,tc1)*x_vec[0] + C_V(c1,tc1)*x_vec[1] +
C_W(c1,tc1)*x_vec[2];
                v1 = C_U(c1,tc1)*y_vec[0] + C_V(c1,tc1)*y_vec[1] +
C_W(c1,tc1)*y_vec[2];
                w1 = C_U(c1,tc1)*z_vec[0] + C_V(c1,tc1)*z_vec[1] +
C_W(c1,tc1)*z_vec[2];

                x[i] = (x0*C_VOLUME(c0,tc0) + x1*C_VOLUME(c1,tc1))
                        / (C_VOLUME(c0,tc0) + C_VOLUME(c1,tc1));
                y[i] = (y0*C_VOLUME(c0,tc0) + y1*C_VOLUME(c1,tc1))
                        / (C_VOLUME(c0,tc0) + C_VOLUME(c1,tc1));

                u[i] = (C_VOLUME(c1,tc1)*u0 + C_VOLUME(c0,tc0)*u1)
                        / (C_VOLUME(c0,tc0) + C_VOLUME(c1,tc1));
                v[i] = (C_VOLUME(c1,tc1)*v0 + C_VOLUME(c0,tc0)*v1)
                        / (C_VOLUME(c0,tc0) + C_VOLUME(c1,tc1));
                w[i] = (C_VOLUME(c1,tc1)*w0 + C_VOLUME(c0,tc0)*w1)
                        / (C_VOLUME(c0,tc0) + C_VOLUME(c1,tc1));
                p[i] = (C_VOLUME(c1,tc1)*C_P(c0,tc0) + C_VOLUME(c0,tc0)*C_P(c1,tc1))
                        / (C_VOLUME(c0,tc0) + C_VOLUME(c1,tc1));
                rho[i] = (C_VOLUME(c1,tc1)*C_R(c0,tc0) + C_VOLUME(c0,tc0)*C_R(c1,tc1))
                        / (C_VOLUME(c0,tc0) + C_VOLUME(c1,tc1));
                b[i] = 0;
                Finish:
                sum+=b[i];
                i++;
        }
        end_f_loop(f,th)


        for(i = 1; i<= n_pts; i++){
                fprintf(ofp8,"%lf %lf %lf %lf %lf %lf %lf %d\n", x[i], y[i],
                        u[i], v[i], w[i], p[i], rho[i], b[i]);
        }
        fclose(ofp8);

        n_st = sum;
        n_norm = n_pts / n_st;

ne1 = (int *)malloc((n_st + 1) * sizeof(int));
        if(ne1 == NULL){
                                (void)fprintf(stderr, "ERROR: Malloc failed ne1");
                                (void)exit(EXIT_FAILURE);    /* or return EXIT_FAILURE; */
                                        }
ne2 = (int *)malloc((n_st + 1) * sizeof(int));
        if(ne2 == NULL){
                                (void)fprintf(stderr, "ERROR: Malloc failed ne2");
                                (void)exit(EXIT_FAILURE);    /* or return EXIT_FAILURE; */
                                        }
x_int = (double *)malloc((n_st+1) * sizeof(double));
        if(x_int == NULL){
                                (void)fprintf(stderr, "ERROR: Malloc failed x_int");
                                (void)exit(EXIT_FAILURE);    /* or return EXIT_FAILURE; */
                                        }
y_int = (double *)malloc((n_st+1) * sizeof(double));
        if(y_int == NULL){
                                (void)fprintf(stderr, "ERROR: Malloc failed y_int");
                                (void)exit(EXIT_FAILURE);    /* or return EXIT_FAILURE; */
                                        }
u_int = (double *)malloc((n_st+1) * sizeof(double));
```

```
        if(u_int == NULL){
                            (void)fprintf(stderr, "ERROR: Malloc failed u_int");
                            (void)exit(EXIT_FAILURE);    /* or return EXIT_FAILURE; */
                                    }
v_int = (double *)malloc((n_st+1) * sizeof(double));
        if(v_int == NULL){
                            (void)fprintf(stderr, "ERROR: Malloc failed v_int");
                            (void)exit(EXIT_FAILURE);    /* or return EXIT_FAILURE; */
                                    }
w_int = (double *)malloc((n_st+1) * sizeof(double));
        if(w_int == NULL){
                            (void)fprintf(stderr, "ERROR: Malloc failed w_int");
                            (void)exit(EXIT_FAILURE);    /* or return EXIT_FAILURE; */
                                    }
p_int = (double *)malloc((n_st+1) * sizeof(double));
        if(p_int == NULL){
                            (void)fprintf(stderr, "ERROR: Malloc failed p_int");
                            (void)exit(EXIT_FAILURE);    /* or return EXIT_FAILURE; */
                                    }
rho_int = (double *)malloc((n_st+1) * sizeof(double));
        if(rho_int == NULL){
                            (void)fprintf(stderr, "ERROR: Malloc failed rho_int");
                            (void)exit(EXIT_FAILURE);    /* or return EXIT_FAILURE; */
                                    }

x_up = (double **)malloc((n_st+1) * sizeof(double *));
        for(i = 0; i < (n_st+1); i++)
                        x_up[i] = (double *)malloc((n_norm+1) * sizeof(double));

        if(x_up == NULL){
            (void)fprintf(stderr, "ERROR: Malloc failed x_up");
            (void)exit(EXIT_FAILURE);    /* or return EXIT_FAILURE; */
                    }
y_up = (double **)malloc((n_st+1) * sizeof(double *));
        for(i = 0; i < (n_st+1); i++)
                        y_up[i] = (double *)malloc((n_norm+1) * sizeof(double));

        if(y_up == NULL){
            (void)fprintf(stderr, "ERROR: Malloc failed y_up");
            (void)exit(EXIT_FAILURE);    /* or return EXIT_FAILURE; */
                    }
u_up = (double **)malloc((n_st+1) * sizeof(double *));
        for(i = 0; i < (n_st+1); i++)
                        u_up[i] = (double *)malloc((n_norm+1) * sizeof(double));

        if(u_up == NULL){
            (void)fprintf(stderr, "ERROR: Malloc failed u_up");
            (void)exit(EXIT_FAILURE);    /* or return EXIT_FAILURE; */
                    }
v_up = (double **)malloc((n_st+1) * sizeof(double *));
        for(i = 0; i < (n_st+1); i++)
                        v_up[i] = (double *)malloc((n_norm+1) * sizeof(double));

        if(v_up == NULL){
            (void)fprintf(stderr, "ERROR: Malloc failed v_up");
            (void)exit(EXIT_FAILURE);    /* or return EXIT_FAILURE; */
                    }
w_up = (double **)malloc((n_st+1) * sizeof(double *));
        for(i = 0; i < (n_st+1); i++)
                        w_up[i] = (double *)malloc((n_norm+1) * sizeof(double));

        if(w_up == NULL){
            (void)fprintf(stderr, "ERROR: Malloc failed w_up");
            (void)exit(EXIT_FAILURE);    /* or return EXIT_FAILURE; */
                    }
p_up = (double **)malloc((n_st+1) * sizeof(double *));
        for(i = 0; i < (n_st+1); i++)
```

```
                        p_up[i] = (double *)malloc((n_norm+1) * sizeof(double));

        if(x_up == NULL){
            (void)fprintf(stderr, "ERROR: Malloc failed x_up");
            (void)exit(EXIT_FAILURE);    /* or return EXIT_FAILURE; */
                 }
rho_up = (double **)malloc((n_st+1) * sizeof(double *));
        for(i = 0; i < (n_st+1); i++)
                        rho_up[i] = (double *)malloc((n_norm+1) * sizeof(double));

        if(rho_up == NULL){
            (void)fprintf(stderr, "ERROR: Malloc failed rho_up");
            (void)exit(EXIT_FAILURE);    /* or return EXIT_FAILURE; */
                 }

x_dn = (double **)malloc((n_st+1) * sizeof(double *));
        for(i = 0; i < (n_st+1); i++)
                        x_dn[i] = (double *)malloc((n_norm+1) * sizeof(double));

        if(x_dn == NULL){
            (void)fprintf(stderr, "ERROR: Malloc failed x_dn");
            (void)exit(EXIT_FAILURE);    /* or return EXIT_FAILURE; */
                 }
y_dn = (double **)malloc((n_st+1) * sizeof(double *));
        for(i = 0; i < (n_st+1); i++)
                        y_dn[i] = (double *)malloc((n_norm+1) * sizeof(double));

        if(y_dn == NULL){
            (void)fprintf(stderr, "ERROR: Malloc failed y_dn");
            (void)exit(EXIT_FAILURE);    /* or return EXIT_FAILURE; */
                 }
u_dn = (double **)malloc((n_st+1) * sizeof(double *));
        for(i = 0; i < (n_st+1); i++)
                        u_dn[i] = (double *)malloc((n_norm+1) * sizeof(double));

        if(u_dn == NULL){
            (void)fprintf(stderr, "ERROR: Malloc failed u_dn");
            (void)exit(EXIT_FAILURE);    /* or return EXIT_FAILURE; */
                 }
v_dn = (double **)malloc((n_st+1) * sizeof(double *));
        for(i = 0; i < (n_st+1); i++)
                        v_dn[i] = (double *)malloc((n_norm+1) * sizeof(double));

        if(v_dn == NULL){
            (void)fprintf(stderr, "ERROR: Malloc failed v_dn");
            (void)exit(EXIT_FAILURE);    /* or return EXIT_FAILURE; */
                 }
w_dn = (double **)malloc((n_st+1) * sizeof(double *));
        for(i = 0; i < (n_st+1); i++)
                        w_dn[i] = (double *)malloc((n_norm+1) * sizeof(double));

        if(w_dn == NULL){
            (void)fprintf(stderr, "ERROR: Malloc failed w_dn");
            (void)exit(EXIT_FAILURE);    /* or return EXIT_FAILURE; */
                 }
p_dn = (double **)malloc((n_st+1) * sizeof(double *));
        for(i = 0; i < (n_st+1); i++)
                        p_dn[i] = (double *)malloc((n_norm+1) * sizeof(double));

        if(x_dn == NULL){
            (void)fprintf(stderr, "ERROR: Malloc failed x_dn");
            (void)exit(EXIT_FAILURE);    /* or return EXIT_FAILURE; */
                 }
rho_dn = (double **)malloc((n_st+1) * sizeof(double *));
        for(i = 0; i < (n_st+1); i++)
                        rho_dn[i] = (double *)malloc((n_norm+1) * sizeof(double));
```

```
    if(rho_dn == NULL){
        (void)fprintf(stderr, "ERROR: Malloc failed rho_dn");
        (void)exit(EXIT_FAILURE);    /* or return EXIT_FAILURE; */
             }

/* get wall points */

n=1;
for(i=1; i<=n_pts;i++){
    if(b[i]==1 && u[i]==0.0){
            x_int[n] = x[i];
            y_int[n] = y[i];
            u_int[n] = u[i];
            v_int[n] = v[i];
            w_int[n] = w[i];
            rho_int[n] = rho[i];
            p_int[n] = p[i];
            n+=1;
    }
    }
/* find stagnation point */

/*stag = 1;*/
del_old = 0.0001;
for(i=1; i < n_st; i++){
    dum1 = p_int[i];
    if(dum1 > del_old){
            del_old = dum1;
            stag = i;
    }
}
/* find leading edge point */


del_old = 100.;
for(i=1; i < n_st; i++){
    dum1 = x_int[i];
    if(dum1 < del_old){
            del_old = dum1;
            LE = i;
    }
}

/*print the stagnation x location*/
stag_x = x_int[stag]-x_int[LE];

/*  shift origin to the leading edge point */
dum1 = x_int[stag];
dum2 = y_int[stag];
for(i=1; i <= n_pts; i++){
    x[i] = x[i] - dum1;
    y[i] = y[i] - dum2;
}
for(i=1; i<=n_st; i++){
    x_int[i] = x_int[i] - dum1;
    y_int[i] = y_int[i] - dum2;
}

/* finds the neighbors of each point */
dum1_old = 0.00001;
for(i=1; i<=n_st; i++){
    dum1 = sqrt((x_int[i]-x_int[stag])*(x_int[i]-x_int[stag])
                        + (y_int[i]-y_int[stag])*(y_int[i]-y_int[stag]));
    if(dum1 > dum1_old){
            dum1_old = dum1;
            tei = i;
    }
```

```
    del_old = 100000.;
    for(j=1; j<=n_st; j++){
            if(j!=i){
                    dum2 = sqrt((x_int[i]-x_int[j])*(x_int[i]-x_int[j])
                                    + (y_int[i]-y_int[j])*(y_int[i]-y_int[j]));
                    ptn = j;
                    if(dum2 < del_old){
                            del_old = dum2;
                            pto = ptn;
                    }
            }
    }
            ne1[i] = pto;

            /*second loop to find runner up*/

            del_old = 100000.;
            for(j=1; j<=n_st; j++){
            if(j!=i && j!=ne1[i]){
                    dum2 = sqrt((x_int[i]-x_int[j])*(x_int[i]-x_int[j])
                                    + (y_int[i]-y_int[j])*(y_int[i]-y_int[j]));
                    ptn = j;
                    if(dum2 < del_old){
                            del_old = dum2;
                            pto = ptn;
                    }
            }
    }
    ne2[i] = pto;
}

chord = sqrt((x_int[LE]-x_int[tei])*(x_int[LE]-x_int[tei])
                    + (y_int[LE]-y_int[tei])*(y_int[LE]-y_int[tei]));

fprintf(ofp3,"%f\n",chord);



/*populates upper surface points*/
dum1 = 0.;
x_up[1][1] = x_int[stag];
y_up[1][1] = y_int[stag];
u_up[1][1] = u_int[stag];
v_up[1][1] = v_int[stag];
w_up[1][1] = w_int[stag];
rho_up[1][1] = rho_int[stag];
p_up[1][1] = p_int[stag];

if(y_int[ne1[stag]] > y_int[ne2[stag]])
    j = ne1[stag];
else
    j = ne2[stag];
x_up[2][1] = x_int[j];
y_up[2][1] = y_int[j];
u_up[2][1] = u_int[j];
v_up[2][1] = v_int[j];
w_up[2][1] = w_int[j];
rho_up[2][1] = rho_int[j];
p_up[2][1] = p_int[j];
i=3;
cond = 0;
sum = 0;
while(cond != 1 && sum < ((n_st+1)*(n_st+2))){
    if(x_int[ne1[j]] == x_up[i-2][1] && y_int[ne1[j]] == y_up[i-2][1])
            j = ne2[j];
    else if(x_int[ne2[j]] == x_up[i-2][1] && y_int[ne2[j]] == y_up[i-2][1])
```

```
            j = ne1[j];
    else{
            (void)fprintf(stderr, "ERROR: lost the upper thread at %d\n",i);
    }


    x_up[i][1] = x_int[j];
     y_up[i][1] = y_int[j];
     u_up[i][1] = u_int[j];
     v_up[i][1] = v_int[j];
     w_up[i][1] = w_int[j];
     rho_up[i][1] = rho_int[j];
     p_up[i][1] = p_int[j];

    dum1 = sqrt((x_up[i][1]-x_up[1][1])*(x_up[i][1]-x_up[1][1])
                            + (y_up[i][1]-y_up[1][1])*(y_up[i][1]-y_up[1][1]));
    if(dum1 > c_cut*chord)
            cond = 1;
    i++;
    sum++;
}
n_upper = i-1;


/*populates lower surface points*/
dum1 = 0;
x_dn[1][1] = x_int[stag];
y_dn[1][1] = y_int[stag];
u_dn[1][1] = u_int[stag];
v_dn[1][1] = v_int[stag];
w_dn[1][1] = w_int[stag];
rho_dn[1][1] = rho_int[stag];
p_dn[1][1] = p_int[stag];

if(y_int[ne1[stag]] > y_int[ne2[stag]])
    j = ne2[stag];
else
    j = ne1[stag];
x_dn[2][1] = x_int[j];
y_dn[2][1] = y_int[j];
u_dn[2][1] = u_int[j];
v_dn[2][1] = v_int[j];
w_dn[2][1] = w_int[j];
rho_dn[2][1] = rho_int[j];
p_dn[2][1] = p_int[j];
i=3;
cond = 0;
sum = 0;
while(cond != 1 && sum < ((n_st+1)*(n_st+2))){
    if(x_int[ne1[j]] == x_dn[i-2][1] && y_int[ne1[j]] == y_dn[i-2][1])
            j = ne2[j];
    else if(x_int[ne2[j]] == x_dn[i-2][1] && y_int[ne2[j]] == y_dn[i-2][1])
            j = ne1[j];
    else{
            (void)fprintf(stderr, "ERROR: lost the lower thread at %d\n",i);
    }


    x_dn[i][1] = x_int[j];
     y_dn[i][1] = y_int[j];
     u_dn[i][1] = u_int[j];
     v_dn[i][1] = v_int[j];
     w_dn[i][1] = w_int[j];
     rho_dn[i][1] = rho_int[j];
     p_dn[i][1] = p_int[j];

    dum1 = sqrt((x_dn[i][1]-x_dn[1][1])*(x_dn[i][1]-x_dn[1][1])
```

```
                              + (y_dn[i][1]-y_dn[1][1])*(y_dn[i][1]-y_dn[1][1]));
        if(dum1 >= c_cut*chord)
                cond = 1;
        i++;
        sum++;
    }
    n_lower = i-1;

    /*  Time for the second layer */

        nx_up = (double *)malloc((n_upper+1) * sizeof(double));
                if(nx_up == NULL){
                            (void)fprintf(stderr, "ERROR: Malloc failed nx_up");
                            (void)exit(EXIT_FAILURE);    /* or return EXIT_FAILURE; */
                                }
        nx_dn = (double *)malloc((n_lower+1) * sizeof(double));
                if(nx_dn == NULL){
                            (void)fprintf(stderr, "ERROR: Malloc failed nx_dn");
                            (void)exit(EXIT_FAILURE);    /* or return EXIT_FAILURE; */
                                }
        ny_up = (double *)malloc((n_upper+1) * sizeof(double));
                if(ny_up == NULL){
                            (void)fprintf(stderr, "ERROR: Malloc failed ny_up");
                            (void)exit(EXIT_FAILURE);    /* or return EXIT_FAILURE; */
                                }
        ny_dn = (double *)malloc((n_lower+1) * sizeof(double));
                if(ny_dn == NULL){
                            (void)fprintf(stderr, "ERROR: Malloc failed ny_dn");
                            (void)exit(EXIT_FAILURE);    /* or return EXIT_FAILURE; */
                                }

    free(x_int);
    free(y_int);
    free(u_int);
    free(v_int);
    free(w_int);
    free(rho_int);
    free(p_int);

    fprintf(ofp2,"lower surface\n");
    fprintf(ofp2,"n_st= %d\n",n_lower);
    fprintf(ofp2,"n_norm= %d\n", norm_st);

    for(i = 1; i <= n_lower; i++){
        fprintf(ofp2,"%lf %lf %lf %lf %lf %lf %lf\n", y_dn[i][1], x_dn[i][1],
v_dn[i][1],
                        u_dn[i][1], w_dn[i][1], p_dn[i][1], rho_dn[i][1]);
    }


    /*
    for(i = 1; i <= n_upper; i++){
        fprintf(ofp,"%lf %lf %lf %lf %lf %lf %lf\n", y_up[i][1], x_up[i][1],
v_up[i][1],
                        u_up[i][1], w_up[i][1], p_up[i][1], rho_up[i][1]);
    }
    */

/****************** Populate arrays in the normal direction ******************/
/*Input: x_up, y_up, u_up, v_up, w_up, p_up, rho_up */
/*Input: x_dn, y_dn, u_dn, v_dn, w_dn, p_dn, rho_dn */
/*Output: same as input */

    for(n=2; n <= norm_st; n++){
        if(n == 2){
                /*      This Part Gets slope for first row */
```

```
                dum1 = ((y_dn[1][1] - y_dn[2][1]) + (y_up[2][1] - y_up[1][1])) / 2;
                dum2 = ((x_dn[1][1] - x_dn[2][1]) + (x_up[2][1] - x_up[1][1])) / 2;
                nx_dn[1] = x_dn[1][1] - dum1/sqrt(dum1*dum1 + dum2*dum2);
                ny_dn[1] = y_dn[1][1] + dum2/sqrt(dum1*dum1 + dum2*dum2);
                nx_up[1] = nx_dn[1];
                ny_up[1] = ny_dn[1];


                for(i=2; i < n_upper; i++){

                dum1 = ((y_up[i+1][1] - y_up[i][1]) + (y_up[i][1] - y_up[i-1][1])) /
2.;
                dum2 = ((x_up[i+1][1] - x_up[i][1]) + (x_up[i][1] - x_up[i-1][1])) /
2.;
                        nx_up[i] = x_up[i][1] - dum1/sqrt(dum1*dum1 + dum2*dum2);
                        ny_up[i] = y_up[i][1] + dum2/sqrt(dum1*dum1 + dum2*dum2);
                }

                for(i=2; i < n_lower; i++){

                dum1 = ((y_dn[i][1] - y_dn[i+1][1]) + (y_dn[i-1][1] - y_dn[i][1])) /
2.;
                dum2 = ((x_dn[i][1] - x_dn[i+1][1]) + (x_dn[i-1][1] - x_dn[i][1])) /
2.;
                        nx_dn[i] = x_dn[i][1] - dum1/sqrt(dum1*dum1 + dum2*dum2);
                        ny_dn[i] = y_dn[i][1] + dum2/sqrt(dum1*dum1 + dum2*dum2);
                }

                dum1 = (y_up[n_upper][1] - y_up[n_upper - 1][1]);
                dum2 = (x_up[n_upper][1] - x_up[n_upper - 1][1]);
                nx_up[n_upper] = x_up[n_upper][1] - dum1/sqrt(dum1*dum1 + dum2*dum2);
                ny_up[n_upper] = y_up[n_upper][1] + dum2/sqrt(dum1*dum1 + dum2*dum2);



                dum1 = (y_dn[n_lower - 1][1] - y_dn[n_lower][1]);
                dum2 = (x_dn[n_lower - 1][1] - x_dn[n_lower][1]);
                nx_dn[n_lower] = x_dn[n_lower][1] - dum1/sqrt(dum1*dum1 + dum2*dum2);
                ny_dn[n_lower] = y_dn[n_lower][1] + dum2/sqrt(dum1*dum1 + dum2*dum2);
        }




    /* finds second layer */


    for(i=1; i<=n_upper; i++){
        fprintf(ofp7,"%lf %lf %lf %lf\n",x_up[i][1],y_up[i][1],nx_up[i], ny_up[i]);
        dum1_old = 100000.0;
        if(n==2){
                nx_up1 = nx_up[i];
                ny_up1 = ny_up[i];
                dum3 = 1.0;
        }
        if(n>=3){
                dum3 = sqrt((x_up[i][n-1] - x_up[i][n-2])*(x_up[i][n-1] - x_up[i][n-
2]) +
                                (y_up[i][n-1] - y_up[i][n-2])*(y_up[i][n-1] -
y_up[i][n-2]));
                nx_up1 = (x_up[i][n-1] - x_up[i][n-2]) / dum3;
                ny_up1 = (y_up[i][n-1] - y_up[i][n-2]) / dum3;
        }
        k = 0;
dotloop1:
```

```
        for(j=1; j<=n_pts; j++){
                del = sqrt((x[j] - x_up[i][n-1])*(x[j] - x_up[i][n-1]) +
                                (y[j] - y_up[i][n-1])*(y[j] - y_up[i][n-1]));
                dum1 = del;
                dum2 = (nx_up1*(x[j] - x_up[i][n-1])/del + ny_up1*(y[j] - y_up[i][n-
1])/del);
                if(dum2 > dot && dum1 < 2.5*dum3 && dum1 < dum1_old){
                        ptn = j;
                        dum1_old = dum1;
                        k = 1;
                }
        }
        if (k!= 1){
                dot -= 0.2;
                if(dot <= 0.)
                        goto exitd1;
                goto dotloop1;
        }
        dot = 0.9;
exitd1:
        x_up[i][n] = x[ptn];
        y_up[i][n] = y[ptn];
        u_up[i][n] = u[ptn];
        v_up[i][n] = v[ptn];
        w_up[i][n] = w[ptn];
        rho_up[i][n] = rho[ptn];
        p_up[i][n] = p[ptn];

    }


    for(i=1; i<=n_lower; i++){
        dum1_old = 100000.0;
        if(n==2){
                nx_dn1 = nx_dn[i];
                ny_dn1 = ny_dn[i];
                dum3 = 1.0;
        }
        if(n>=3){
                dum3 = sqrt((x_dn[i][n-1] - x_dn[i][n-2])*(x_dn[i][n-1] - x_dn[i][n-
2]) +
                                (y_dn[i][n-1] - y_dn[i][n-2])*(y_dn[i][n-1] -
y_dn[i][n-2]));
                nx_dn1 = (x_dn[i][n-1] - x_dn[i][n-2]) / dum3;
                ny_dn1 = (y_dn[i][n-1] - y_dn[i][n-2]) / dum3;
        }
        k = 0;
dotloop2:
        for(j=1; j<=n_pts; j++){
                del = sqrt((x[j] - x_dn[i][n-1])*(x[j] - x_dn[i][n-1]) +
                                (y[j] - y_dn[i][n-1])*(y[j] - y_dn[i][n-1]));
                dum1 = del;
                dum2 =(nx_dn1*(x[j] - x_dn[i][n-1])/del + ny_dn1*(y[j] - y_dn[i][n-
1])/del);
                if(dum2 > dot && dum1 < 2.5*dum3 && dum1 < dum1_old){
                        ptn = j;
                        dum1_old = dum1;
                        k = 1;
                }
        }
        if (k!= 1){
                dot -= 0.2;
                if(dot <= 0.)
                        goto exitd2;
                goto dotloop2;
        }
        dot = 0.9;
```

```
exitd2:
        x_dn[i][n] = x[ptn];
        y_dn[i][n] = y[ptn];
        u_dn[i][n] = u[ptn];
        v_dn[i][n] = v[ptn];
        w_dn[i][n] = w[ptn];
        rho_dn[i][n] = rho[ptn];
        p_dn[i][n] = p[ptn];

    }



} /* end n loop */

/********************************** End Populate arrays
*************************************/

    fprintf(ofp9,"upper surface\n");
    fprintf(ofp9,"n_st= %d\n",n_upper);
    fprintf(ofp9,"n_norm= %d\n", norm_st);
    for(n=1; n<=n_upper; n++){
        for(i=1; i<=norm_st; i++){
               fprintf(ofp9,"%lf %lf %lf %lf %lf %lf %lf %lf\n", x_up[n][i],
y_up[n][i], 1.0, u_up[n][i],
                     v_up[n][i], w_up[n][i], p_up[n][i], rho_up[n][i]);

        }
    }


    free(u);
    free(v);
    free(w);
    free(rho);
    free(p);

/*******  SPLINE FITTING ***********/

/************** INPUT ***************/
/* up variables, dn variables, n_upper, n_norm */

/************** OUTPUT **************/
/*  up variables transformed to be orthogonal */

        x_int2 = (double **)malloc((n_upper+1) * sizeof(double *));
               for(i = 0; i < (n_upper+1); i++)
                        x_int2[i] = (double *)malloc((n_norm+1) *
sizeof(double));

               if(x_int2 == NULL){
                   (void)fprintf(stderr, "ERROR: Malloc failed x_int2");
                   (void)exit(EXIT_FAILURE);    /* or return EXIT_FAILURE; */
                 }
        y_int2 = (double **)malloc((n_upper+1) * sizeof(double *));
               for(i = 0; i < (n_upper+1); i++)
                        y_int2[i] = (double *)malloc((n_norm+1) *
sizeof(double));

               if(y_int2 == NULL){
                   (void)fprintf(stderr, "ERROR: Malloc failed y_int2");
                   (void)exit(EXIT_FAILURE);    /* or return EXIT_FAILURE; */
                 }


for(j = 2 ; j <= norm_st ; j++){
        for(i = 1 ; i <= n_upper ; i++){
```

```
        if(j == 2){
        x_int2[i][1] = x_up[i][1];
        y_int2[i][1] = y_up[i][1];
        }

                l = i-1;
                m = i;
                n = i;
                o = i+1;


                Redo:
                if(l < 0){
                dum1 = sqrt((x_dn[1-l][j] - x_up[i][1])*(x_dn[1-l][j] - x_up[i][1]) +
                                (y_dn[1-l][j] - y_up[i][1])*(y_dn[1-l][j] -
y_up[i][1]));
                dum2 = sqrt((x_dn[1-m][j] - x_up[i][1])*(x_dn[1-m][j] - x_up[i][1]) +
                                (y_dn[1-m][j] - y_up[i][1])*(y_dn[1-m][j] -
y_up[i][1]));
                dum3 = sqrt((x_dn[n][j] - x_up[i][1])*(x_dn[n][j] - x_up[i][1]) +
                                (y_dn[n][j] - y_up[i][1])*(y_dn[n][j] - y_up[i][1]));
                dum4 = sqrt((x_dn[o][j] - x_up[i][1])*(x_dn[o][j] - x_up[i][1]) +
                                (y_dn[o][j] - y_up[i][1])*(y_dn[o][j] - y_up[i][1]));
                        v1x = (x_dn[2-l][j] - x_up[i][1]) / dum1;
                        v1y = (y_dn[2-l][j] - y_up[i][1]) / dum1;
                        v2x = (x_dn[2-m][j] - x_up[i][1]) / dum2;
                        v2y = (y_dn[2-m][j] - y_up[i][1]) / dum2;
                        v3x = (x_up[n][j] - x_up[i][1]) / dum3;
                        v3y = (y_up[n][j] - y_up[i][1]) / dum3;
                        v4x = (x_up[o][j] - x_up[i][1]) / dum4;
                        v4y = (y_up[o][j] - y_up[i][1]) / dum4;
                }
                else if(l == 0){
                        dum1 = sqrt((x_dn[2][j] - x_up[i][1])*(x_dn[2][j] -
x_up[i][1]) +
                                        (y_dn[2][j] - y_up[i][1])*(y_dn[2][j] -
y_up[i][1]));
                        dum2 = sqrt((x_up[m][j] - x_up[i][1])*(x_up[m][j] -
x_up[i][1]) +
                                        (y_up[m][j] - y_up[i][1])*(y_up[m][j] -
y_up[i][1]));
                        dum3 = sqrt((x_up[n][j] - x_up[i][1])*(x_up[n][j] -
x_up[i][1]) +
                                        (y_up[n][j] - y_up[i][1])*(y_up[n][j] -
y_up[i][1]));
                        dum4 = sqrt((x_up[o][j] - x_up[i][1])*(x_up[o][j] -
x_up[i][1]) +
                                        (y_up[o][j] - y_up[i][1])*(y_up[o][j] -
y_up[i][1]));
                        v1x = (x_dn[2][j] - x_up[i][1]) / dum1;
                        v1y = (y_dn[2][j] - y_up[i][1]) / dum1;
                        v2x = (x_up[m][j] - x_up[i][1]) / dum2;
                        v2y = (y_up[m][j] - y_up[i][1]) / dum2;
                        v3x = (x_up[n][j] - x_up[i][1]) / dum3;
                        v3y = (y_up[n][j] - y_up[i][1]) / dum3;
                        v4x = (x_up[o][j] - x_up[i][1]) / dum4;
                        v4y = (y_up[o][j] - y_up[i][1]) / dum4;
                }
                else{
                dum1 = sqrt((x_up[l][j] - x_up[i][1])*(x_up[l][j] - x_up[i][1]) +
                                (y_up[l][j] - y_up[i][1])*(y_up[l][j] - y_up[i][1]));
                dum2 = sqrt((x_up[m][j] - x_up[i][1])*(x_up[m][j] - x_up[i][1]) +
                                (y_up[m][j] - y_up[i][1])*(y_up[m][j] - y_up[i][1]));
                dum3 = sqrt((x_up[n][j] - x_up[i][1])*(x_up[n][j] - x_up[i][1]) +
                                (y_up[n][j] - y_up[i][1])*(y_up[n][j] - y_up[i][1]));
                dum4 = sqrt((x_up[o][j] - x_up[i][1])*(x_up[o][j] - x_up[i][1]) +
                                (y_up[o][j] - y_up[i][1])*(y_up[o][j] - y_up[i][1]));
```

```
                v1x = (x_up[l][j] - x_up[i][1]) / dum1;
                v1y = (y_up[l][j] - y_up[i][1]) / dum1;
                v2x = (x_up[m][j] - x_up[i][1]) / dum2;
                v2y = (y_up[m][j] - y_up[i][1]) / dum2;
                v3x = (x_up[n][j] - x_up[i][1]) / dum3;
                v3y = (y_up[n][j] - y_up[i][1]) / dum3;
                v4x = (x_up[o][j] - x_up[i][1]) / dum4;
                v4y = (y_up[o][j] - y_up[i][1]) / dum4;

                }

                th1 = theta_find(v1y,v1x);
                th2 = theta_find(v2y,v2x);
                th3 = theta_find(v3y,v3x);
                th4 = theta_find(v4y,v4x);
                th_norm = theta_find(ny_up[i],nx_up[i]);

    if((th1 >= th_norm && th2 <= th_norm) || (th1 <= th_norm && th2 >= th_norm))
                        pt = l;
    else if((th3 >= th_norm && th4 <= th_norm) || (th3 <= th_norm && th4 >= th_norm))
                        pt = o;

    else{
                l--;
                m--;
                n++;
                o++;
        if(o > n_upper || -l > (n_lower-20)){
                printf("AR too high at st: %d\n",i);
                        l = i-1;
                        m = i;
                        n = i;
                        o = i+1;
                        pt = l;
                        goto Punt;
                    }
                        goto Redo;
                }
                Punt:
                if(i == 1){
                        if(pt > 0){
                        k_end = n_upper - pt + 1;
                        u = (double *)malloc((k_end+1) * sizeof(double));
                                if(u == NULL){
                                    (void)fprintf(stderr, "ERROR: Malloc failed u");
                                    (void)exit(EXIT_FAILURE);   /* or return
EXIT_FAILURE; */
                                         }

                        v = (double *)malloc((k_end+1) * sizeof(double));
                                if(v == NULL){
                                    (void)fprintf(stderr, "ERROR: Malloc failed v");
                                    (void)exit(EXIT_FAILURE);   /* or return
EXIT_FAILURE; */
                                         }

                        w = (double *)malloc((k_end+1) * sizeof(double));
                                if(w == NULL){
                                    (void)fprintf(stderr, "ERROR: Malloc failed w");
                                    (void)exit(EXIT_FAILURE);   /* or return
EXIT_FAILURE; */
                                         }

                        rho = (double *)malloc((k_end+1) * sizeof(double));
                                if(rho == NULL){
                                    (void)fprintf(stderr, "ERROR: Malloc failed rho");
```

```
                                (void)exit(EXIT_FAILURE);   /* or return
EXIT_FAILURE; */
                                     }

                           p = (double *)malloc((k_end+1) * sizeof(double));
                                if(p == NULL){
                                    (void)fprintf(stderr, "ERROR: Malloc failed p");
                                    (void)exit(EXIT_FAILURE);   /* or return
EXIT_FAILURE; */
                                     }

                           ypp = (double *)malloc((k_end+1) * sizeof(double));
                                if(ypp == NULL){
                                    (void)fprintf(stderr, "ERROR: Malloc failed ypp");
                                    (void)exit(EXIT_FAILURE);   /* or return
EXIT_FAILURE; */
                                     }

                           s = (double *)malloc((k_end+1) * sizeof(double));
                                if(s == NULL){
                                    (void)fprintf(stderr, "ERROR: Malloc failed s");
                                    (void)exit(EXIT_FAILURE);   /* or return
EXIT_FAILURE; */
                                     }
                                u[1] = u_up[pt][j];
                                v[1] = v_up[pt][j];
                                w[1] = w_up[pt][j];
                                p[1] = p_up[pt][j];
                                rho[1] = rho_up[pt][j];
                                s[1] = 0.;
                                for(k=2; k <= k_end; k++){
                                s[k] = s[k-1] + sqrt((x_up[k+pt-1][j] - x_up[k+pt-
2][j])*
                                                     (x_up[k+pt-1][j] -
x_up[k+pt-2][j]) +
                                                     (y_up[k+pt-1][j] -
y_up[k+pt-2][j])*
                                                     (y_up[k+pt-1][j] -
y_up[k+pt-2][j]));
                                          u[k] = u_up[k+pt-1][j];
                                          v[k] = v_up[k+pt-1][j];
                                          w[k] = w_up[k+pt-1][j];
                                          p[k] = p_up[k+pt-1][j];
                                          rho[k] = rho_up[k+pt-1][j];
                                }
                           }
                           else if(pt <= 0){
                           k_end = n_upper - pt + 1;
                           u = (double *)malloc((k_end+1) * sizeof(double));
                                if(u == NULL){
                                    (void)fprintf(stderr, "ERROR: Malloc failed u");
                                    (void)exit(EXIT_FAILURE);   /* or return
EXIT_FAILURE; */
                                     }

                           v = (double *)malloc((k_end+1) * sizeof(double));
                                if(v == NULL){
                                    (void)fprintf(stderr, "ERROR: Malloc failed v");
                                    (void)exit(EXIT_FAILURE);   /* or return
EXIT_FAILURE; */
                                     }
                           w = (double *)malloc((k_end+1) * sizeof(double));
                                if(w == NULL){
                                    (void)fprintf(stderr, "ERROR: Malloc failed w");
                                    (void)exit(EXIT_FAILURE);   /* or return
EXIT_FAILURE; */
                                     }
```

```
                    rho = (double *)malloc((k_end+1) * sizeof(double));
                        if(rho == NULL){
                            (void)fprintf(stderr, "ERROR: Malloc failed rho");
                            (void)exit(EXIT_FAILURE);   /* or return
EXIT_FAILURE; */
                                 }

                    p = (double *)malloc((k_end+1) * sizeof(double));
                        if(p == NULL){
                            (void)fprintf(stderr, "ERROR: Malloc failed p");
                            (void)exit(EXIT_FAILURE);   /* or return
EXIT_FAILURE; */
                                 }

                    ypp = (double *)malloc((k_end+1) * sizeof(double));
                        if(ypp == NULL){
                            (void)fprintf(stderr, "ERROR: Malloc failed ypp");
                            (void)exit(EXIT_FAILURE);   /* or return
EXIT_FAILURE; */
                                 }

                    s = (double *)malloc((k_end+1) * sizeof(double));
                        if(s == NULL){
                            (void)fprintf(stderr, "ERROR: Malloc failed s");
                            (void)exit(EXIT_FAILURE);   /* or return
EXIT_FAILURE; */
                                 }
                               u[1] = u_dn[2-pt][j];
                               v[1] = v_dn[2-pt][j];
                               w[1] = w_dn[2-pt][j];
                               p[1] = p_dn[2-pt][j];
                               rho[1] = rho_dn[2-pt][j];
                               s[i] = 0.;
                        for(k=2; k <= k_end; k++){
                        if(k <= 2 - pt){
                        s[k] = s[k-1] + sqrt((x_dn[4-k-pt][j] - x_dn[3-k-
pt][j])*
                                                (x_dn[4-k-pt][j] -
x_dn[3-k-pt][j])+
                                                (y_dn[4-k-pt][j] -
y_dn[3-k-pt][j])*
                                                (y_dn[4-k-pt][j] -
y_dn[3-k-pt][j]));
                                       u[k] = u_dn[3-k-pt][j];
                                       v[k] = v_dn[3-k-pt][j];
                                       w[k] = w_dn[3-k-pt][j];
                                       p[k] = p_dn[3-k-pt][j];
                                       rho[k] = rho_dn[3-k-pt][j];
                               }
                        else{
                        s[k] = s[k-1] + sqrt((x_up[k+pt-1][j] - x_up[k+pt-
2][j])*
                                                (x_up[k+pt-1][j] -
x_up[k+pt-2][j])+
                                                (y_up[k+pt-1][j] -
y_up[k+pt-2][j])*
                                                (y_up[k+pt-1][j] -
y_up[k+pt-2][j]));
                                       u[k] = u_up[k+pt-1][j];
                                       v[k] = v_up[k+pt-1][j];
                                       w[k] = w_up[k+pt-1][j];
                                       p[k] = p_up[k+pt-1][j];
                                       rho[k] = rho_up[k+pt-1][j];
                               }
                               /*if(j == 2)*/
                               /*fprintf(ofp4,"%d %lf %lf\n",k, s[k], 1.0);*/
```

```
                        }
                }

        }


        /* find the point of intersection between the grid line and normal
line */
        if(pt == l){
                x0 = v1x*dum1 + x_up[i][1];
                y0 = v1y*dum1 + y_up[i][1];
                x1 = v2x*dum2 + x_up[i][1];
                y1 = v2y*dum2 + y_up[i][1];
        }
        else if(pt == o){
                x0 = v3x*dum3 + x_up[i][1];
                y0 = v3y*dum3 + y_up[i][1];
                x1 = v4x*dum4 + x_up[i][1];
                y1 = v4y*dum4 + y_up[i][1];
        }
        if(fabs(nx_up[i]) < 1.e-10){
                mg = (y1 - y0) / (x1 - x0);
                x_int2[i][j] = x_up[i][1];
                y_int2[i][j] = y0 - mg*x0 + mg*x_int2[i][j];
        }
        else{
                mn = (ny_up[i] - y_up[i][1]) / (nx_up[i] - x_up[i][1]);
                mg = (y1 - y0) / (x1 - x0);
                x_int2[i][j] = ((y0 - mg*x0) - (y_up[i][1] - mn*x_up[i][1])) /
(mn - mg);
                y_int2[i][j] = y_up[i][1] - mn*x_up[i][1] + mn*x_int2[i][j];
        }
}  /* end the inner i loop */

ypp1 = 0.;
yppn = 0.;
/* spline fit the u */
my_spline(s, u, k_end, ypp, ypp1, yppn);
for(i=1;i<=n_upper;i++){
        bob = sqrt((x_up[i][j] - x_int2[i][j])*(x_up[i][j] - x_int2[i][j]) +
                        (y_up[i][j] - y_int2[i][j])*(y_up[i][j] -
y_int2[i][j]));
        s_p = s[k_end - n_upper + i] + bob;
        u_up[i][j] = my_splint(s, u, ypp, k_end, s_p);
        /* fprintf(ofp4,"%lf %lf\n", s_p, u_up[i][j]); */
}
/* spline fit the v */
my_spline(s, v, k_end, ypp, ypp1, yppn);
for(i=1;i<=n_upper;i++){
        bob = sqrt((x_up[i][j] - x_int2[i][j])*(x_up[i][j] - x_int2[i][j]) +
                        (y_up[i][j] - y_int2[i][j])*(y_up[i][j] -
y_int2[i][j]));
        s_p = s[k_end - n_upper + i] + bob;
        v_up[i][j] = my_splint(s, v, ypp, k_end, s_p);
}
/*spline fit the w */
my_spline(s, w, k_end, ypp, ypp1, yppn);
for(i=1;i<=n_upper;i++){
        bob = sqrt((x_up[i][j] - x_int2[i][j])*(x_up[i][j] - x_int2[i][j]) +
                        (y_up[i][j] - y_int2[i][j])*(y_up[i][j] -
y_int2[i][j]));
        s_p = s[k_end - n_upper + i] + bob;
        w_up[i][j] = my_splint(s, w, ypp, k_end, s_p);
}
/*spline fit the p */
my_spline(s, p, k_end, ypp, ypp1, yppn);
for(i=1;i<=n_upper;i++){
```

```
                bob = sqrt((x_up[i][j] - x_int2[i][j])*(x_up[i][j] - x_int2[i][j]) +
                                (y_up[i][j] - y_int2[i][j])*(y_up[i][j] -
y_int2[i][j]));
                s_p = s[k_end - n_upper + i] + bob;
                p_up[i][j] = my_splint(s, p, ypp, k_end, s_p);
        }
        /*spline fit the rho */
        my_spline(s, rho, k_end, ypp, ypp1, yppn);
        for(i=1;i<=n_upper;i++){
                bob = sqrt((x_up[i][j] - x_int2[i][j])*(x_up[i][j] - x_int2[i][j]) +
                                (y_up[i][j] - y_int2[i][j])*(y_up[i][j] -
y_int2[i][j]));
                s_p = s[k_end - n_upper + i] + bob;
                rho_up[i][j] = my_splint(s, rho, ypp, k_end, s_p);
        }
        free(u);
        free(v);
        free(w);
        free(p);
        free(rho);
        free(ypp);
        free(s);
                }  /* end the outer j loop */

/******  END OF THE CIRCUMFERENTIAL SPLINE ********/

/******   NOW WE SPLINE IN THE NORMAL DIRECTION ******/



s = (double *)malloc((norm_st+1) * sizeof(double));
        if(s == NULL){
                (void)fprintf(stderr, "ERROR: Malloc failed s");
                (void)exit(EXIT_FAILURE);    /* or return EXIT_FAILURE; */
        }
s_th = (double *)malloc((stab_j+1) * sizeof(double));
        if(s_th == NULL){
                (void)fprintf(stderr, "ERROR: Malloc failed s_th");
                (void)exit(EXIT_FAILURE);    /* or return EXIT_FAILURE; */
        }
cut = (int *)malloc((n_upper+1) * sizeof(int));
        if(cut == NULL){
                (void)fprintf(stderr, "ERROR: Malloc failed cut");
                (void)exit(EXIT_FAILURE);    /* or return EXIT_FAILURE; */
        }
ypp = (double *)malloc((norm_st+1) * sizeof(double));
        if(ypp == NULL){
                (void)fprintf(stderr, "ERROR: Malloc failed ypp");
                (void)exit(EXIT_FAILURE);    /* or return EXIT_FAILURE; */
        }
u = (double *)malloc((norm_st+1) * sizeof(double));
       if(u == NULL){
                (void)fprintf(stderr, "ERROR: Malloc failed u");
                (void)exit(EXIT_FAILURE);    /* or return EXIT_FAILURE; */
        }
v = (double *)malloc((norm_st+1) * sizeof(double));
        if(v == NULL){
                (void)fprintf(stderr, "ERROR: Malloc failed v");
                (void)exit(EXIT_FAILURE);    /* or return EXIT_FAILURE; */
        }
w = (double *)malloc((norm_st+1) * sizeof(double));
        if(w == NULL){
                (void)fprintf(stderr, "ERROR: Malloc failed w");
                (void)exit(EXIT_FAILURE);    /* or return EXIT_FAILURE; */
        }
p = (double *)malloc((norm_st+1) * sizeof(double));
        if(p == NULL){
```

```
                (void)fprintf(stderr, "ERROR: Malloc failed p");
                (void)exit(EXIT_FAILURE);    /* or return EXIT_FAILURE; */
        }
rho = (double *)malloc((norm_st+1) * sizeof(double));
        if(rho == NULL){
                (void)fprintf(stderr, "ERROR: Malloc failed rho");
                (void)exit(EXIT_FAILURE);    /* or return EXIT_FAILURE; */
        }
x_sp = (double **)malloc((n_upper+1) * sizeof(double *));
        for(i = 0; i < (n_upper+1); i++)
                x_sp[i] = (double *)malloc((stab_j+1) * sizeof(double));

        if(x_sp == NULL){
            (void)fprintf(stderr, "ERROR: Malloc failed x_sp");
            (void)exit(EXIT_FAILURE);    /* or return EXIT_FAILURE; */
        }
x_sp = (double **)malloc((n_upper+1) * sizeof(double *));
        for(i = 0; i < (n_upper+1); i++)
                x_sp[i] = (double *)malloc((stab_j+1) * sizeof(double));

        if(x_sp == NULL){
            (void)fprintf(stderr, "ERROR: Malloc failed x_sp");
            (void)exit(EXIT_FAILURE);    /* or return EXIT_FAILURE; */
        }
y_sp = (double **)malloc((n_upper+1) * sizeof(double *));
        for(i = 0; i < (n_upper+1); i++)
                y_sp[i] = (double *)malloc((stab_j+1) * sizeof(double));

        if(y_sp == NULL){
            (void)fprintf(stderr, "ERROR: Malloc failed y_sp");
            (void)exit(EXIT_FAILURE);    /* or return EXIT_FAILURE; */
        }
u_sp = (double **)malloc((n_upper+1) * sizeof(double *));
        for(i = 0; i < (n_upper+1); i++)
                u_sp[i] = (double *)malloc((stab_j+1) * sizeof(double));

        if(u_sp == NULL){
            (void)fprintf(stderr, "ERROR: Malloc failed u_sp");
            (void)exit(EXIT_FAILURE);    /* or return EXIT_FAILURE; */
        }
v_sp = (double **)malloc((n_upper+1) * sizeof(double *));
        for(i = 0; i < (n_upper+1); i++)
                v_sp[i] = (double *)malloc((stab_j+1) * sizeof(double));

        if(v_sp == NULL){
            (void)fprintf(stderr, "ERROR: Malloc failed v_sp");
            (void)exit(EXIT_FAILURE);    /* or return EXIT_FAILURE; */
        }
w_sp = (double **)malloc((n_upper+1) * sizeof(double *));
        for(i = 0; i < (n_upper+1); i++)
                w_sp[i] = (double *)malloc((stab_j+1) * sizeof(double));

        if(w_sp == NULL){
            (void)fprintf(stderr, "ERROR: Malloc failed w_sp");
            (void)exit(EXIT_FAILURE);    /* or return EXIT_FAILURE; */
        }
p_sp = (double **)malloc((n_upper+1) * sizeof(double *));
        for(i = 0; i < (n_upper+1); i++)
                p_sp[i] = (double *)malloc((stab_j+1) * sizeof(double));

        if(p_sp == NULL){
            (void)fprintf(stderr, "ERROR: Malloc failed p_sp");
            (void)exit(EXIT_FAILURE);    /* or return EXIT_FAILURE; */
        }
rho_sp = (double **)malloc((n_upper+1) * sizeof(double *));
        for(i = 0; i < (n_upper+1); i++)
                rho_sp[i] = (double *)malloc((stab_j+1) * sizeof(double));
```

```
        if(rho_sp == NULL){
            (void)fprintf(stderr, "ERROR: Malloc failed rho_sp");
            (void)exit(EXIT_FAILURE);    /* or return EXIT_FAILURE; */
        }




/* find the BL top */
for(i=1; i<= n_upper; i++){
        cut[i] = norm_st;
        for(j = 2; j<= norm_st; j++){
                v_tot = sqrt(u_up[i][j]*u_up[i][j] + v_up[i][j]*v_up[i][j] +
w_up[i][j]*w_up[i][j]);
                v_tot_old = sqrt(u_up[i][j-1]*u_up[i][j-1] + v_up[i][j-1]*v_up[i][j-1]
                                + w_up[i][j-1]*w_up[i][j-1]);
                dum1 = (v_tot - v_tot_old) / v_tot;
                if(fabs(dum1) <= 0.02 || dum1 < 0){
                        cut[i] = j;
                        goto getout;
                }
        }

getout: dum1 = 1.;

}


d_theta = Pi / 2. / stab_j;
ypp1 = 0.;
yppn = 0.;
for(i = 1; i <= n_upper; i++){
        /*scale = 1.5;*/
rescale:
        theta = 0.;
        x_sp[i][1] = x_int2[i][1];
        y_sp[i][1] = y_int2[i][1];
        dum1 = sqrt((y_int2[i][cut[i]] - y_int2[i][1])*(y_int2[i][cut[i]] -
y_int2[i][1])
                + (x_int2[i][cut[i]] - x_int2[i][1])*(x_int2[i][cut[i]] -
x_int2[i][1]));
        dum2 = (x_int2[i][cut[i]] - x_int2[i][1]) / dum1;
        dum3 = (y_int2[i][cut[i]] - y_int2[i][1]) / dum1;
        ang = theta_find(dum3,dum2);
        /*fprintf(ofp8, "%d %lf %lf %lf\n", i, dum2, dum1, ang);*/
        for(j = 2; j <= stab_j; j++){
                theta += d_theta;
                s_th[j] = (1 - cos(theta))*dum1*scale;

                if(ang > Pi/2. && ang <= Pi){
                        x_sp[i][j] = x_up[i][1] - s_th[j]*cos(Pi - ang);
                        y_sp[i][j] = y_up[i][1] + s_th[j]*sin(Pi - ang);
                }
                else if(ang > Pi && ang <= 3.*Pi/2.){
                        x_sp[i][j] = x_up[i][1] - s_th[j]*cos(ang - Pi);
                        y_sp[i][j] = y_up[i][1] - s_th[j]*sin(ang - Pi);
                }
                else if(ang <= Pi){
                        x_sp[i][j] = x_up[i][1] + s_th[j]*cos(ang);
                        y_sp[i][j] = y_up[i][1] + s_th[j]*sin(ang);
                }
        }

        s[1] = 0.;
        u[1] = u_up[i][1];
```

```
        v[1] = v_up[i][1];
        w[1] = w_up[i][1];
        rho[1] = rho_up[i][1];
        p[1] = p_up[i][1];
        for(j = 2; j <= norm_st; j++){
                s[j] = sqrt((x_int2[i][j] - x_int2[i][1])*(x_int2[i][j] -
x_int2[i][1]) +
                                (y_int2[i][j] - y_int2[i][1])*(y_int2[i][j] -
y_int2[i][1]));
                u[j] = u_up[i][j];
                v[j] = v_up[i][j];
                w[j] = w_up[i][j];
                rho[j] = rho_up[i][j];
                p[j] = p_up[i][j];
        }

        if(s[norm_st] < s_th[stab_j]){
                scale -= 0.1;
                goto rescale;
        }
         /*
        if(i == 73){
                fprintf(ofp7,"before_spline\n");
                for(j = 1; j<=norm_st; j++){
                        fprintf(ofp7,"%lf %lf\n",s[j],u[j]);
                }
        }
        */
        my_spline(s,u,norm_st,ypp,ypp1,yppn);
        for(j = 1; j<=stab_j; j++){
                u_sp[i][j] = my_splint(s, u, ypp, norm_st, s_th[j]);
        }
        /*
        if(i == 73){
                 fprintf(ofp7,"after_spline\n");
                 for(j = 1; j<=stab_j; j++){
                         fprintf(ofp7,"%lf %lf\n",s_th[j],u_sp[i][j]);
                 }
         }
        */
        my_spline(s,v,norm_st,ypp,ypp1,yppn);
        for(j = 1; j<=stab_j; j++){
                v_sp[i][j] = my_splint(s, v, ypp, norm_st, s_th[j]);
        }

        my_spline(s,w,norm_st,ypp,ypp1,yppn);
        for(j = 1; j<=stab_j; j++){
                 w_sp[i][j] = my_splint(s, w, ypp, norm_st, s_th[j]);
         }

        my_spline(s,rho,norm_st,ypp,ypp1,yppn);
         for(j = 1; j<=stab_j; j++){
                 rho_sp[i][j] = my_splint(s, rho, ypp, norm_st, s_th[j]);
         }

        my_spline(s,p,norm_st,ypp,ypp1,yppn);
         for(j = 1; j<=stab_j; j++){
                 p_sp[i][j] = my_splint(s, p, ypp, norm_st, s_th[j]);
         }

/*fprintf(ofp6, "%d %lf\n",i,scale);*/

}
free(u);
free(v);
free(w);
free(rho);
```

```
free(p);
free(s);
free(s_th);

/*****  END OF NORMAL DIRECTION SPLINE *****/



/* transform things to be "normal" to leading edge */
x_vec[0] = cos(sw_ang*Pi/180.);
x_vec[1] = 0;
x_vec[2] = sin(sw_ang*Pi/180.);
y_vec[0] = 0.;
y_vec[1] = 1.;
y_vec[2] = 0.;
z_vec[0] = -sin(sw_ang*Pi/180.);
z_vec[1] = 0.;
z_vec[2] = cos(sw_ang*Pi/180.);

    fprintf(ofp,"upper surface\n");
    fprintf(ofp,"n_st= %d\n",n_upper);
    fprintf(ofp,"n_norm= %d\n", norm_st);
    for(n=1; n<=n_upper; n++){
        for(i=1; i<=norm_st; i++){
                /*
                x_dum = -y_up[n][i];
                y_dum = -x_up[n][i];
                u_dum = -v_up[n][i];
                v_dum = -u_up[n][i];
                x_up[n][i] = x_dum;
                y_up[n][i] = y_dum;
                u_up[n][i] = u_dum;
                v_up[n][i] = v_dum;
                w_up[n][i] = -w_up[n][i];
                */
                if(i >= 2){
                        x_up[n][i] = x_int2[n][i];
                        y_up[n][i] = y_int2[n][i];
                }
                fprintf(ofp,"%lf %lf %lf %lf %lf %lf %lf %lf\n", x_up[n][i],
y_up[n][i], 1.0, u_up[n][i],
                                v_up[n][i], w_up[n][i], p_up[n][i], rho_up[n][i]);

        }
    }


        /* now lets print the normal spline fit */
        fprintf(ofp5,"%d\n",n_upper);
        fprintf(ofp5,"n_st= %d\n",stab_j);
         fprintf(ofp5,"n_norm= %lf\n", chord);
        fprintf(ofp6,"%d\n",n_upper);
         fprintf(ofp6,"n_st= %d\n",stab_j);
         fprintf(ofp6,"n_norm= %lf\n", chord);
        for(n=1; n<=n_upper; n++){
                for(i=1; i<=stab_j; i++){
                        if(i>1){
                                y_dum = sqrt((y_sp[n][i] - y_sp[n][1])*(y_sp[n][i] -
y_sp[n][1]) +
                                                (x_sp[n][i] - x_sp[n][1])*(x_sp[n][i] -
x_sp[n][1]));
                                u_dum = u_sp[n][i]*ny_up[n] - v_sp[n][i]*nx_up[n];
                                v_dum = u_sp[n][i]*nx_up[n] + v_sp[n][i]*ny_up[n];
                        }
                        else {
                                y_dum = 0.0;
                                u_dum = u_sp[n][i];
```

```
                        v_dum = v_sp[n][i];
                    }
                    /* print flattened xc lastrac grid */
                    fprintf(ofp5,"%lf %lf %lf %lf %lf %lf %lf %lf\n", x_sp[n][i],
y_dum, 1.0, u_dum,
                        v_dum, w_sp[n][i], p_sp[n][i], rho_sp[n][i]);
                    /*  print actual grid */
                    fprintf(ofp6,"%lf %lf %lf %lf %lf %lf %lf %lf\n", x_sp[n][i],
y_sp[n][i], 1.0, u_dum,
                        v_dum, w_sp[n][i], p_sp[n][i], rho_sp[n][i]);
            }
        }


    x_dum = 0;
    fprintf(ofp4,"%i\n",n_upper);
    fprintf(ofp4,"%i\n",norm_st);
    fprintf(ofp4,"%lf\n",chord);
    fprintf(ofp3,"%d %lf\n",n_upper,stag_x);
        for(n=1; n<=n_upper; n++){
            if(n>1){
            x_dum += sqrt((y_up[n][1] - y_up[n-1][1])*(y_up[n][1] - y_up[n-1][1])
+
                        (x_up[n][1] - x_up[n-1][1])*(x_up[n][1] - x_up[n-
1][1]));
            }
            else
                x_dum = 0;

        /* compute the radius of curvature for lastrac */
        if(n == 1){  /*stencil shifts to accomodate left boundary*/
             /* first derivative of y w.r.t. x */
            dum1 = (-3.0*y_up[n][1] + 4.*y_up[n+1][1] - 1.0*y_up[n+2][1]) /
                ((x_up[n+1][1] - x_up[n][1]) + (x_up[n+2][1] - x_up[n+1][1]));
            /* second derivative of y w.r.t. x */
            dum2 = (y_up[n][1] - 2.0*y_up[n+1][1] + y_up[n+2][1]) /
                ((x_up[n+2][1] - x_up[n+1][1])*(x_up[n+1][1] - x_up[n][1]));
            dum3 = 1.0 + dum1*dum1;     /* numerator argument for radius of curv.
*/
            Rc = pow(dum3,3./2.) / fabs(dum2);  /* Radius of curvature */

            }
            else if (n == n_upper){  /*setencil shifts to accomoadate right
boundary*/
            /* first derivative of y w.r.t. x */
            dum1 = (-3.0*y_up[n][1] + 4.*y_up[n-1][1] - 1.0*y_up[n-2][1]) /
                ((x_up[n][1] - x_up[n-1][1] ) + (x_up[n-1][1] - x_up[n-
2][1]));
            /* second derivative of y w.r.t. x */
            dum2 = (y_up[n][1] - 2.0*y_up[n-1][1] + y_up[n-2][1]) /
                ((x_up[n-1][1] - x_up[n-2][1])*(x_up[n][1] - x_up[n-1][1]));
            /* numerator argument for radius of curv. */
            dum3 = 1.0 + dum1*dum1;
            Rc = pow(dum3,3./2.) / fabs(dum2);  /* Radius of curvature */
            }
            else{  /*symmetric stencil for interior points*/
                 /* first derivative of y w.r.t. x */
            dum1 = (y_up[n+1][1] - y_up[n-1][1]) / ((x_up[n+1][1] - x_up[n][1]) +
                (x_up[n][1] - x_up[n-1][1]));
            /* second derivative */
            dum2 = (y_up[n-1][1] - 2.0*y_up[n][1] + y_up[n+1][1]) /
                ((y_up[n+1][1] - y_up[n][1])*(y_up[n][1] - y_up[n-1][1]));
            dum3 = 1.0 + dum1*dum1;     /* numerator argument for radius of curv.
*/
            Rc = pow(dum3,3./2.) / fabs(dum2);  /* Radius of curvature */

            }
```

```
                fprintf(ofp3,"%d %lf\n",norm_st,Rc);

                for(i=1; i <= norm_st ; i++){
                        if(i>1){
                        y_dum = sqrt((y_up[n][i] - y_up[n][1])*(y_up[n][i] -
y_up[n][1]) +
                                        (x_up[n][i] - x_up[n][1])*(x_up[n][i] -
x_up[n][1]));
                        u_dum = u_up[n][i]*ny_up[n] - v_up[n][i]*nx_up[n];
                        v_dum = u_up[n][i]*nx_up[n] + v_up[n][i]*ny_up[n];
                        }
                        else{
                                y_dum = 0;
                                u_dum = 0;
                                v_dum = 0;
                        }
                        z_dum = 1.;
                        w_dum = w_up[n][i];
                        x0 = x_dum;
                        y0 = y_dum;
                        u0 = x_vec[0]*u_dum + x_vec[1]*v_dum + x_vec[2]*w_dum;
                        v0 = y_vec[0]*u_dum + y_vec[1]*v_dum + y_vec[2]*w_dum;
                        w0 = z_vec[0]*u_dum + z_vec[1]*v_dum + z_vec[2]*w_dum;

                /* ofp3 prints the body fitted geometry output before normal spline
fit */
                fprintf(ofp3,"%lf %lf %lf %lf %lf %lf %lf %lf\n", x0, y0, z_dum, u0,
                                v0, w0, p_up[n][i], rho_up[n][i]);
                /* for further analysis, print data with x aligned with freestream and
                        y being normal to the surface */
                fprintf(ofp4,"%lf %lf %lf %lf %lf %lf %lf %lf\n", x_up[n][i], y0,
                        z_dum, u_up[n][i],
                                v_up[n][i], w_up[n][i], p_up[n][i], rho_up[n][i]);
                }
        }

        free(x_up);
        free(y_up);
        free(u_up);
        free(v_up);
        free(w_up);
        free(p_up);
        free(rho_up);
        free(x_int2);
        free(y_int2);
        free(nx_up);
        free(ny_up);
        free(x_dn);
        free(y_dn);
        free(u_dn);
        free(v_dn);
        free(w_dn);
        free(p_dn);
        free(rho_dn);
        free(nx_dn);
        free(ny_dn);
        free(x_sp);
        free(y_sp);
        free(u_sp);
        free(v_sp);
        free(w_sp);
        free(rho_sp);
        free(p_sp);
        fclose(ofp);
        fclose(ofp2);
        fclose(ofp3);
```

```
        fclose(ofp4);
        fclose(ofp5);
        fclose(ofp6);
        fclose(ofp7);
        fclose(ofp9);
}  /*  END OF MAIN */
```

FUNCTIONS

```
void my_spline(double *x_sp, double *y_sp, int nl, double *ypp, double ypp1, double
yppn){
/*      Spline takes in 1-D arrays (pointers) x and y, as well as the first
        and last values of the second derivative, ypp1 and yppn, and constructs
        the 1-D array (pointer) that is ypp[1..N].
*/
        int i,j,k;
        double *alp, *bet, *gam, *rhs;
        double  m;
        FILE *ofpp;

        ofpp = fopen("spline_deb.txt", "w");

        fprintf(ofpp,"the spline is being called\n");

        alp = (double *)malloc((nl + 1) * sizeof(double));
                if(alp == NULL){
                    (void)fprintf(stderr, "ERROR: Malloc failed u");
                    (void)exit(EXIT_FAILURE);    /* or return EXIT_FAILURE; */
                        }
        bet = (double *)malloc((nl + 1) * sizeof(double));
                if(bet == NULL){
                    (void)fprintf(stderr, "ERROR: Malloc failed u");
                    (void)exit(EXIT_FAILURE);    /* or return EXIT_FAILURE; */
                         }
        gam = (double *)malloc((nl + 1) * sizeof(double));
                if(gam == NULL){
                    (void)fprintf(stderr, "ERROR: Malloc failed u");
                    (void)exit(EXIT_FAILURE);    /* or return EXIT_FAILURE; */
                         }

        rhs = (double *)malloc((nl + 1) * sizeof(double));
                if(rhs == NULL){
                    (void)fprintf(stderr, "ERROR: Malloc failed u");
                    (void)exit(EXIT_FAILURE);    /* or return EXIT_FAILURE; */
                         }
        ypp[1] = ypp1;
        ypp[nl] = yppn;
        m = x_sp[2];
        alp[2] = (x_sp[2] - x_sp[1]) / 6.;
        rhs[2] = -alp[2]*ypp[1] + ((y_sp[3] - y_sp[2]) / (x_sp[3] - x_sp[2]) -
(y_sp[2] - y_sp[1]) / (x_sp[2] - x_sp[1]));
        alp[nl - 1] = (x_sp[nl - 1] - x_sp[nl - 2]) / 6.;
        rhs[nl - 1] = -alp[nl - 1]*ypp[nl] + ((y_sp[nl] - y_sp[nl - 1])
                        / (x_sp[nl] - x_sp[nl - 1]) - (y_sp[nl - 1] - y_sp[nl - 2]) /
(x_sp[nl - 1] - x_sp[nl - 2]));

        /* populates the alpha, beta, and gamma coefficients for tri-diag matrix */
        /* that solves the system for ypp */

        for(j = 2;j < nl;j++){
                if(j > 2)alp[j] = (x_sp[j] - x_sp[j-1]) / 6.;
                bet[j] = (x_sp[j+1] - x_sp[j-1]) / 3.;
                if(j < nl - 1)gam[j] = (x_sp[j+1] - x_sp[j]) / 6.;

                if(j > 2 && j < nl-1){
```

```
                        rhs[j] = (y_sp[j+1] - y_sp[j]) / (x_sp[j+1] - x_sp[j]) -
(y_sp[j] - y_sp[j-1]) / (x_sp[j] - x_sp[j-1]);
                }
        }

        /* forward elimination of the thomas algorithm */
        for(k=3; k < nl; k++){
                m = alp[k]/bet[k-1];
                bet[k] = bet[k] - m*gam[k-1];
                rhs[k] = rhs[k] - m*rhs[k-1];
        }
        /* backward substitution of the thomas algorithm */
        ypp[nl-1] = rhs[nl-1] / bet[nl - 1];
        for(k=nl - 2; k > 1; k--){
                ypp[k] = (rhs[k] - gam[k]*ypp[k+1]) / bet[k];
        }


        free(alp);
        free(bet);
        free(gam);
        free(rhs);
        fclose(ofpp);
}

double my_splint(double *x_sp, double *y_sp, double *ypp, int nl, double x){

        int klo,khi,k;
        double h,b,a,ret;
        FILE *ofppp;

        ofppp = fopen("splint_deb.txt","w");

        /*fprintf(ofppp,"the splint is being called\n");*/

        klo=1;
        khi=nl;
        while (khi-klo > 1) {
                k=(khi+klo) >> 1;
                if (x_sp[k] > x) khi=k;
                else klo=k;
                }
        h=x_sp[khi]-x_sp[klo];
        if (h == 0.0) fprintf(ofppp,"Bad x_sp input to routine splint\n");
        a=(x_sp[khi]-x)/h;
        b=(x-x_sp[klo])/h;
        ret=a*y_sp[klo]+b*y_sp[khi]+((a*a*a-a)*ypp[klo]+(b*b*b-
b)*ypp[khi])*(h*h)/6.0;
        fclose(ofppp);
        return ret;
}


#define Pi 3.14159

double theta_find(double dely, double delx){
        /* double dely, delx; */
        double theta;

        if(dely >= 0. && delx >= 0.)
                theta = atan(dely/delx);
        else if(dely >= 0. && delx < 0)
                theta = Pi - atan(-dely/delx);
        else if(dely < 0. && delx >= 0.)
                theta = -atan(-dely/delx);  /* was 2*pi - atan() */
        else if(dely < 0. && delx < 0.)
                theta = Pi + atan(dely/delx);
```

```
        return theta;

}

double ang_find(double dely, double delx){
        /* double dely, delx; */
        double theta;

        if(dely >= 0. && delx >= 0.)
                theta = atan(dely/delx);
        else if(dely >= 0. && delx < 0)
                theta = Pi - atan(-dely/delx);
        else if(dely < 0. && delx >= 0.)
                theta = -atan(-dely/delx);  /* was 2*pi - atan() */
        else if(dely < 0. && delx < 0.)
                theta = Pi + atan(dely/delx);
        return theta;

}
```

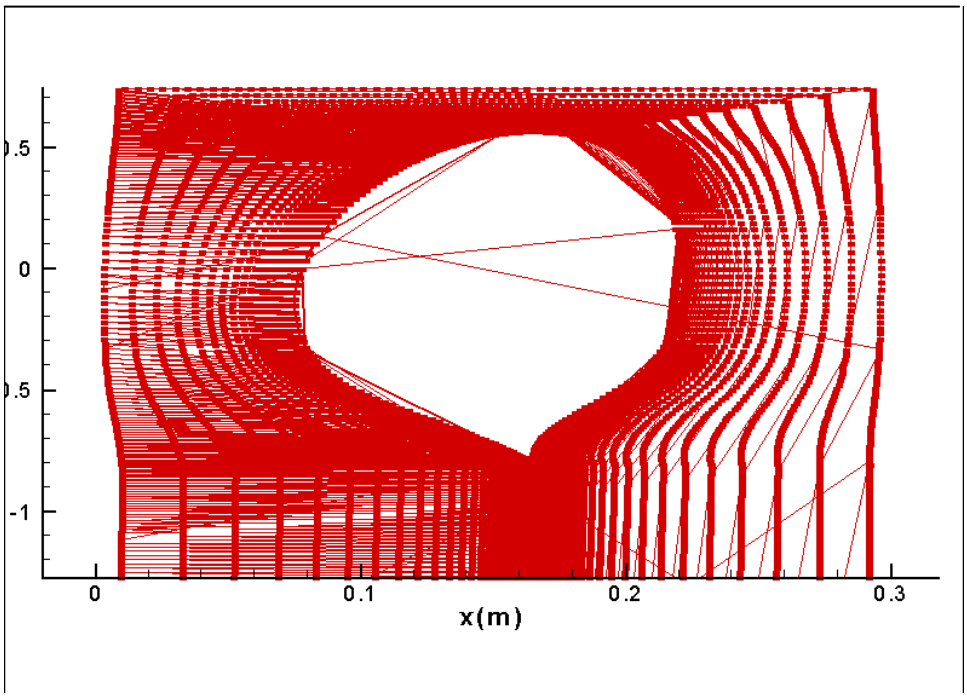


Fig. A - 1 Unordered data set extracted from FLUENT$_{TM}$.

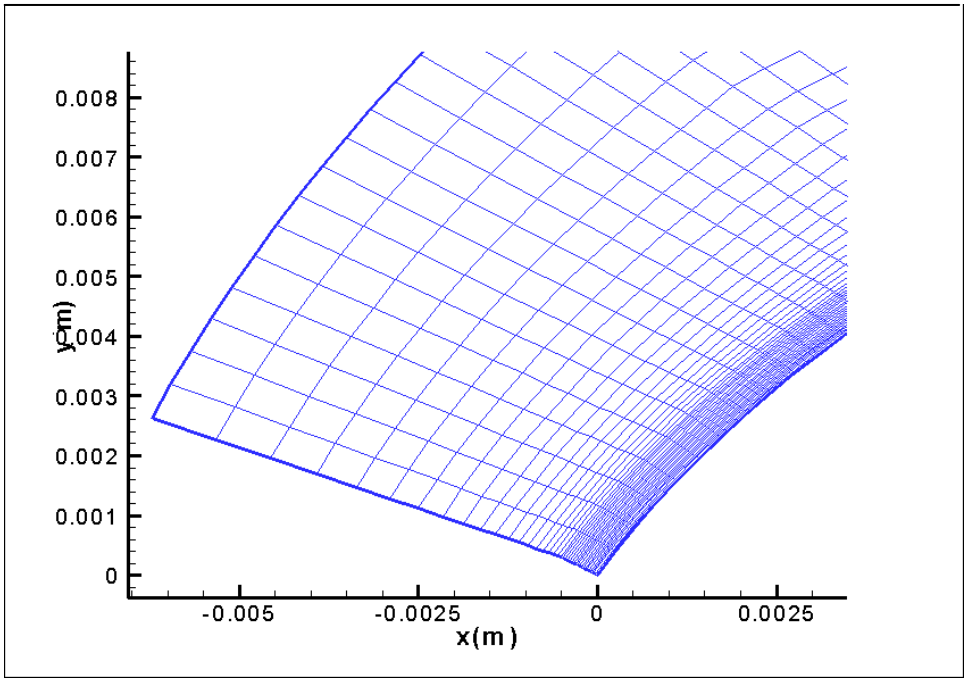


Fig. A - 2 Ordered data set extracted from FLUENT$_{TM}$.

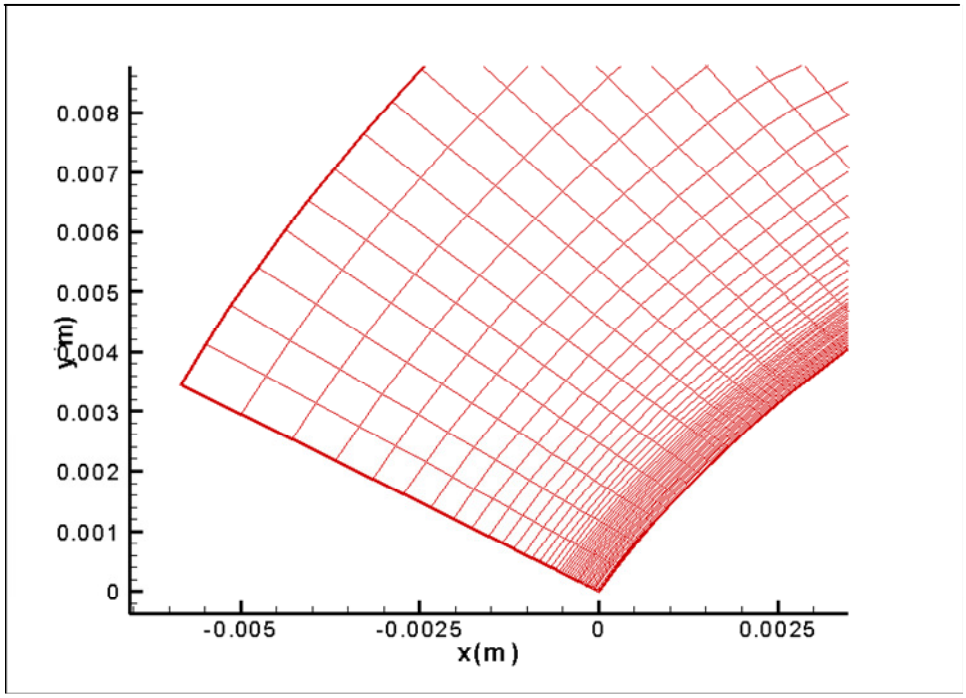


Fig. A - 3 Application of orthogonal correction to ordered data set extracted from FLUENT$_{TM}$.

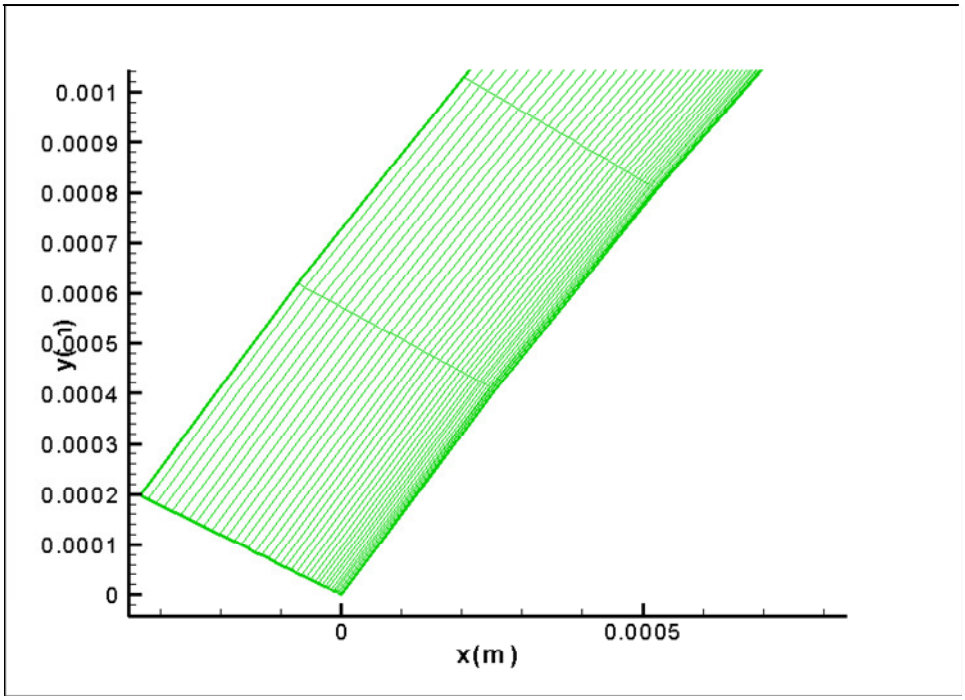


Fig. A - 4 Application of normal cubic spline clustering to orthogonal data set extracted from FLUENT$_{TM}$.

VITA

Name:	Richard George Rhodes
Address:	Texas A&M University, Department of Aerospace Engineering H.R. Bright Building, Rm. 701, Ross Street – TAMU 3141 College Station TX 77843-3141
Email Address:	rickrho@gmail.com
Education:	B.S., Aerospace Engineering, Texas A&M University, 2005 M.S., Aerospace Engineering, Texas A&M University, 2008